NTNU
Norwegian University of
Science and Technology

Compression in XML search engines

Ola Natvig

Master of Science in Computer Science
Submission date: June 2010
Supervisor: Svein Erik Bratsberg, IDI
Co-supervisor: Øystein Torbjørnsen, Fast, a Microsoft® subsidiary

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Examine how compression techniques may be used efficiently in XML search engines. Focus should be on the following aspects: Compressed index size and query performance.

Assignment given: 15. January 2010
Supervisor: Svein Erik Bratsberg, IDI

Abstract

The structure of XML documents can be used by search engines to answer structured queries or to provide better relevancy. Several index structures exist for search in XML data. This study focuses on inverted lists with dictionary coded path types and dewey coded path instances. The dewey coded path index is large, but could be compressed. This study examines query processing with indexes encoded using well known integer coding methods *VByte* and *PFor(delta)* and methods tailored for the dewey index.

Intersection queries and structural queries are evaluated. In addition to standard document level skipping, skip operations for path types are implemented and evaluated. Four extensions over plain *PFor* methods are proposed and tested. *Path type sorting* sorts dewey codes on their path types and store all deweys from one path type together. *Column wise* dewey storage stores the deweys in columns instead of rows. *Prefix coding* a well known method, is adapted to the column wise dewey storage, and *dynamic column wise* method chooses between row wise and column wise storage based on the compressed data.

Experiments are performed on a XML collection based on Wikipedia. Queries are generated from the TREC 06 efficiency task query trace. Several different types of structural queries have been executed.

Experiments show that column wise methods perform good on both intersection and structural queries. The dynamic column wise scheme is in most cases the best, and creates the smallest index. Special purpose skipping for path types makes some queries extremely fast and can be implemented with only limited storage footprint. The performance of in-memory search with multi-threaded query execution is limited by memory bandwidth.

Preface

This is a master thesis for the Master in Computer Technology program at the Department of Computer and Information Science at the Norwegian University of Science and Technology. The work have been performed during the spring of 2010.

I like to thank my supervisors; Svein Erik Bratsberg and Øystein Torbjørnsen. Their ideas and feedback have been very valuable. I would also like to thank *Microsoft® corporation* for lending me hardware for experiments.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Searching in large collections is a common task in peoples day to day life. Common to most types of search is a *query language* and results *ranked* based on some measure of *relevance*. Several models for providing ranked results to different types of queries exist. Textbook models such as the *vector model* and the *probabilistic model* are examples of such methods, where each document is interpreted as a *bag of words*[BYRN99].

Interpreting documents as bags of words is a simple, yet quite effective way of providing a basis for answering the queries specified from the user. Bag of words indexes can also be constructed with quite low storage costs[MRS09a]. However, the bag of word view of a document removes a lot of the structure which might be very important in order to calculate good ranks for the documents. Also, modern users may not want to retrieve *documents* any longer, perhaps the granularity of searches is changing. Modern users might want to get only the relevant parts of documents.

XML is one way to represent structured and semi structured data. It is a quite popular one, and XML is for instance used in document formats for many text processors. By indexing XML instead of plain text, one can include structural information into the index.

While the text-book bag of words models are quite space economical, storing XML information in the index is not. The focus of this thesis will be on the storage and compression schemes for the XML structural information included in the index and how queries against these representations can be executed efficiently. The index representation assumed in this thesis will be described in more detail in Section 1.2.

This is a master thesis in the course *TDT4900 - Computer and Information Science, Master Thesis* spring, 2010. The task has been to investigate compression techniques for inverted indexes in a XML search system. The work has been performed

in cooperation with *Microsoft® Corporation* which have provided the initial idea to the task, example data collections and hardware for experiments to be run on. This work extends on a pre project done in the fall 2009, this will be discussed more in depth in Section 1.3.

1.1 Background and motivation

As described previously in this chapter, advanced query features require more complex index structures. In this section some of the most important sources of inspiration for this thesis will be mentioned. In general one could say that this thesis is an experimental study of how to adapt some of the most common inverted index compression techniques to XML indexes. Therefore the main background will be the field of inverted index compression and query processing, XML search engines and also implementation details.

1.1.1 Compressed inverted indexes

The uncompressed inverted index for document collections, will consume about 40% of the size of the document collection[MRS09a]. Since collections often are big, inverted indexes can become very big. A large index will require a large number of hard drives which in turn is expensive. Standard inverted list compression method can reduce the storage requirements for a given index by as much as 75%[MRS09a].

Saving disk space is a good thing, but there are several other reasons for compressing the inverted index. If the index consumes less space then it is also faster to load that index from disk during query processing. This way, spending extra CPU-cycles to decompress the index can make query processing faster, since the disk otherwise would be the limiting device.

Modern search engines caches data in main memory. This data can be results of frequent queries, the inverted lists for frequently used terms or parts of these lists (probably all of the three)[ZLS08]. Compression greatly affects the amount of data which can reside in such main memory caches, making it possible to answer many queries without ever accessing the disk.

In the future there will probably be search engines keeping all; or most of its index in main memory. Index compression will most certainly be a key technology in enabling this. Modern computers have several types of memory, ranging from the very slow hard drive to the extremely fast registers. The CPU on a modern computer can have several megabytes of on-chip super fast cache, compressed data can also lead to better hit ratios for such caches.

1.1.2 High throughput and lightweight compression

Compression involves a trade off between the compression ratio obtained, and the amount of computation required to decode (and encode) each value. Much recent research has focused on obtaining very fast decompression, while sacrificing some of the compression ratio. The fastest of these methods achieves decoding rates of several gigabytes per second [ZHNB06], while still retaining acceptable compression. Common to many of these high throughput methods is that the architecture of modern CPUs and memory systems are taken into account.

The key to many of the fastest available compression schemes is that they group values together before compressing them. This way, the algorithms can decode multiple values at once, possibly even a fixed number of values at once, making it easier for the implementor to write the code as optimal as possible for the one case where there are for instance 32 values to be decoded.

1.1.3 Managed programming languages

Most prior experimental studies on compression have been performed using programming languages such as C and C++ which is compiled into native machine code. Several modern search engines are built using newer programming languages, such as Java and Microsoft .NET. Especially Java has seen a large number of open source search engine implementations. Lucene, probably the best known open source search engine was originally developed using Java and there exists a Lucene implementation in .NET. Terrier [OAP+06] and MG4J [Vig09] are other examples of Java search engines.

Programs written for runtimes such as Java and Microsoft .NET is not compiled into machine executable code, instead they are compiled into an intermediate code which is executed on top of a virtual machine. These virtual machines supply memory management through garbage collection, and just-in-time compilation of intermediate code to machine executable code. The performance implications of both garbage collection and just-in-time compilation are hard to predict prior to execution. This might be the reason why most experimental studies employ the more predictable statically compiled languages.

There exist methods for the virtual machine languages to invoke statically compiled programs. One might suggest doing this in performance critical parts of programs to leverage the more predictable and probable higher performance of programming languages such as C and C++. However, such "out-of-runtime" calls will come at a higher cost than regular method invocations, and this might offset the performance gains. However, when looking at such techniques in combination with the new very fast compression schemes which compresses multiple values at the time one might amortize each such expensive "out-of-runtime" call over multiple decoded values.

1.2 Index structure

Since there are several possible ways to organize XML indexes, one specific way is selected as the case for this thesis. This organization is based on additional XML-scope information encoded in a normal word-level inverted index. In addition to occurrence positions, one XML-scope field is added to each occurrence. The scope field consists of two parts, the first part is the *path type id*, the path type id uniquely defines the sequence of labels representing the scope of the occurrence. The other part is a *dewey* code which specifies the position of the occurrence within the document. The dewey code can be seen as the instance of the scope, while the path type is the type of the scope. An early adoption of the dewey scheme can be found in [TVB+02].

All the unique paths in the collection is kept in an array and can quickly be accessed by the path type identifier. One way to assign path type identifiers is to assign the identifiers in the order of which the pat types are seen during indexing. If the collection is homogeneous, the most common path types will be observed early, and therefore get assigned lower path type identifiers. An example of a index with path types and dewey codes are shown in Figure 1.1. This is the type of index which is assumed in this thesis, and the concrete version is adapted from [Gri07].

<document>	1
Text	1.1(.1)
<list>	1.2
<elem>E1</elem>	1.2.1(.1)
<elem>E2</elem>	1.2.2(.1)
<elem>	1.2.3
<big>Big</big>	1.2.3.1(.1)
</elem>	
<elem>	1.2.4
Text	1.2.4.1(.1)
...	

(a) Document with Dewey order

1	document
2	document/p
3	document/list
4	document/list/elem
5	document/list/elem/big
6	document/list/elem/small

E1	$(2, 4, 1.2.1.1)$
E2	$(3, 4, 1.2.2.1)$
Big	$(4, 5, 1.2.3.1.1)$
Text	$(1, 2, 1.1.1), (5, 6, 1.2.4.1.1)$

(b) Unique paths and a word level inverted file. Index entires of the form $((pos_i, path\_type_i, dewey_i)^+)$

Figure 1.1: Simple scope index example

The dewey code of a scope is defined as $d_p.c$ where d_p is the dewey code of the parent scope, and c is the ordinal position of the scope within the parent scope. If for instance a scope 1.2.1 has four children, the second child node has the dewey code 1.2.1.2. This means that if two occurrences have some common ancestor in the document structure, they will share some prefix of their dewey codes.

This index structure makes it possible to see the path type of each occurrence, and the position of the occurrence within the structure of the XML document. However, since only the scope of the single occurrence itself is coded in the index, the information regarding the full structure of the document is not available.

Using this structure makes it easy to for instance select occurrences from a specific set of path types. One can compare occurrences in different inverted lists to check if they are within the same scope. Some of the specific use cases that will be evaluated in the thesis will be described in detail in Chapter 2.

1.3 Prior work

In the fall of 2009 the author worked with similar problems in the course *TDT4590 - Complex Computer Systems, Specialization Project*. This preliminary work included a study of the state of the art in index compression. Also experiments were conducted with a focus on compressed index size, index build time and time to decode the index. Some methods to compress or improve compression of XML indexes were suggested and tested. The project report [Nat09] contains detailed descriptions of the experiments and the results.

Some portions of this thesis will include material produced for the project report in [Nat09]. Either in a form close to the original, or in a more summarizing manner. Specifically, the background information on index compression in Section 3.3 and 3.4 will be very similar to that found in [Nat09]. The path type sorting method (Section 4.2.6), and the column-wise dewey encoding (Section 4.2.3 were also introduced and tested in [Nat09]. Last, the results presented in Section 7.2 are the results produced in [Nat09].

This master thesis aims to extend the work done in [Nat09] by focusing on query processing. Also this thesis will offer a more in depth description of the implementation details, as well as the architecture of the *NETing minimal XML search engine*.

1.4 Readers guide

The remainder of this thesis is structured as following, Chapter 2 contains a more in depth explanation of the problem statement. In Chapter 3 an introduction to the theoretical background of the field of index compression and query processing towards compressed indexes will be given. Possible methods and ideas for both compression schemes and query processing techniques will be proposed in Chapter 4, and an introduction of the solutions actually implemented will be given in Chapter 5. In Chapter 6 the set of experiments needed for the experimental study

will be described and Chapter 7 will describe the results of these experiments. In addition a brief description of the results from [Nat09] can be found in Section 7.2. The concluding remarks and ideas for further work is located in Chapter 8.

Chapter 2

Problem definition

While XML query languages such as *XPath* and *XQuery* are very rich. The main focus of this study is the compressed index representation and not XML-query processing in general. A few key query operations towards the compressed inverted index will be selected for further experimentation. In the following sections a description of why these operations are important tasks in answering more complex queries will be given.

The XML scope information in the index may not necessarily be used to answer XML queries specified in XPath or XQuery. They can be used as input to ranking algorithms or perhaps to enhance phrase search. In a search engine these use cases could be equally important as many of the more exotic features present in rich query languages.

2.1 Path selection

An XPath expression selects sets of nodes from XML documents [Cla99]. A special syntax is used to specify the (possible partial) path from the root of a document to the nodes matching the expression. The path expressions look very much like *URL*s or paths in the file system hierarchy.

XPath expressions are divided into steps. Each step can contain two or three elements. An axis, a node test and an optional predicate. XPath defines several *axes*. Each axis encodes a relationship in the XML-tree. All XPath axes are shown in Table 2.1. The syntax for a XPath step is `axis::node-test[predicate]`.

In compact syntax, the child axis is often written as `/` and the descendant axis as `//`. The node test filters the nodes returned from the axis on their label. For

Axis	Description
`child`	Selects all nodes which is the children of the current node.
`parent`	Selects all nodes which is the parent of the current node.
`ancestor`	Selects all nodes which is the ancestor (parent, grandparent etc..) of the current node.
`ancestor-or-self`	Same as `ancestor` but result includes the current node as well.
`decendant`	Selects all nodes which is the descendant (child, grandchild etc..) of the current node.
`decendant-or-self`	Same as `decendant` but result includes the current node as well.
`following`	Selects all nodes after the closing tag of the current node.
`following-sibling`	Selects all sibling (shares same parent) nodes after the current node.
`preceding`	Selects all nodes before the start tag of the current node.
`preceding-sibling`	Selects all sibling nodes before the current node.
`self`	Selects the current node.
`attribute`	Selects the attributes of the current node.
`namespace`	Selects all namespaces of the current node.

Table 2.1: Description of XPath axes.

instance an expression `child::paragraph` will only match child nodes with the label "paragraph". Node tests can be on the type of node also. One could for instance select all text nodes which is the children of the current node with the expression `child::text()`.

The last and optional "predicate" applies extra filtering, based on the nodes returned. XPath defines several functions usable in predicates. Predicates can test both the content of the nodes and the position of the nodes (for instance only select the first child node).

For a more in depth explanation of XPath, the XPath tutorial at `http://www.w3schools.com/XPath/` could be consulted.

2.2 Path selection in search engines

Different parts of documents have different importance for users. A uses which searches for any given topic will most certainly be more satisfied if the results returned describes the topic in document header, or perhaps in bold face text. Search engines can employ zone-scoring in order to give various document parts more importance[MRS09b].

Path selection can be used to search parts of the documents, one could for instance search only headers. XPath supports selecting the first n child elements of a given node, this could also allow the search engine to search in perhaps the three first paragraphs of each document. The inclusion of such fine grained access methods could be used in advanced ranking algorithms, or it could be exposed to the users through some advanced search interface.

2.3 Structural containment search

When scoring multi-term queries the intersection of the different inverted lists will need to be calculated[MRS09c]. In some search systems word positions are indexed in order to answer "phrase-search" queries. When using word position, phrases might not be real phrases. Phrases with consecutive word positions might span document structures. With XML scopes in the index, one can specify phrase queries that require the phrase to be contained within the same XML-scope for instance the same paragraph.

Such a phrase query should be specified as the phrase and an XPath expression which selects the valid scopes. The paths of the words matching the queries should be descendants of the XPath expression in the query. If the phrase should be contained within the `//paragraph` it could also contain postings from for instance `paragraph/bold` scopes as long as the words share the same `paragraph` node.

Searching for words in the same scope are useful even when no phrase requirements are specified. Users might be interested in documents where a set of words appear in the same scope. One could for instance search for "Pink Floyd" and "Dark Side of The Moon" in the same paragraph. Or perhaps "achilles " and "injury" in the same sentence. Proximity based search (comparing word position) might be able to achieve somewhat similar results, however, without attention to the structure of the document.

2.4 Important features in the XML index

Assuming the index structure described in the introduction (Section 1.2). Many XPath expressions can be evaluated by only looking on the path types. The only XPath features which cannot be answered by analyzing path types are the predicate parts of the expressions.

Some XPath predicates can be applied in a post processing step. Others, such as the `last()` predicate are difficult to answer, since the index does not contain information about the cardinalities of the different XML-nodes. To answer such queries extra info need to be encoded. However, execution of XPath in general is

beyond the scope of this thesis.

Structural containment will also require effective path type filtering. Occurrences with a scope prefixed by the containment paths will be candidates for the results. In addition auxiliary data structures describing which path types that are prefixes of others will be needed. Two word occurrences can only be within the same scope if their path types share a common prefix long enough to identify the containment scope. Also, the dewey code for each of the identifiers need to share a prefix of the same length. This means that one will need efficient mechanisms to compare prefixes of dewey codes.

These two operations, *path type filtering* and *dewey prefix compraison* will probably be key functions in XML query evaluation. The compressed representation of the XML index will be a key enabling factor for supplying these features in a speedy manner. Compression affects both the amount of data to be transfered between disk, main memory and the CPU. Compression also affect the computational cost of query operations. Either for the better if query components can operate efficiently on the compressed representation, or for the worse if the data need to be decompressed first.

Additionally, an index representation which allow query processors to disregard some portion of the data during processing might result in more efficient processing. One such method often refereed to in the literature is skipping [MZ96, MRS09d, SC07]. Skipping will be discussed in Section 3.5.

Even if the index is to be used to answer XML-aware queries the basic functionality for calculating inverted list intersections need to be present. It might still be that for many structural queries the intersection of a document level index data will be the most demanding task.

2.5 Concrete problem statement

The focus of this thesis is XML index compression and the interactions between XML index compression and query processing. Both theoretical and experimental analysis should be performed in order to propose the best index representation. Due to limitations in available time, only a small subset of the query types described in the preceding sections can be evaluated.

In order to assess the index representations ability to answer regular queries the performance of AND queries should be measured and commented on. Regular information retrieval tasks often include both intersections and unions of inverted lists. For instance the *continue*-strategy for answering ranked queries proposed in [MZ96] consists of two phases where the first phase is similar to a union (OR) of some of the terms in the query. The second phase calculates the intersection (AND) between the results of the OR phase and the remaining terms. In general OR-

queries are more computational intensive to evaluate since there are more results. However, when focusing on compression; AND-queries are more interesting since the information encoded in skip pointers can be used to speed up the execution of the queries[MZ96].

Experimenting with structural containment queries will provide insight into how well different compression methods handle list intersection, path type filtering and dewey prefix comparison. Different algorithms for evaluating such queries should also be examined.

For structural containment queries it might be that different algorithms suit some queries better than other. For instance, if occurrences within a very common scope such as the `paragraph` scope should be processed different from queries matching a more exotic scope such as a `company` scope.

Experiments should be performed on both well known methods and proposed new solutions. The implementations should run on the Microsoft .NET framework, and should preferably be implemented in the C# programming language. It could, however, for performance reasons be interesting to implement certain portions in C or C++. The possible gains from "going native" should be evaluated.

Chapter 3

Theoretical background

The field of compression in general is a large field. In this chapter an introduction to the state of the art compression methods used in inverted lists will be provided. Also, an introduction to skipping as an optimization available to query processing will be given.

3.1 Shannon's information theory

In 1948 Claude Elwood Shannon published the article "A mathematical Theory of Communication" [Sha48]. This article lays out a mathematical framework for establishing bounds compression and reliability of communication channels. Shannon introduces *Entropy* as a lower bound for compression. Shannon's definition of entropy for a collection of n symbols each with the probability p_i is shown in Equation 3.1.

$$H = -\sum_{i}^{n} p_i \log(p_i) \tag{3.1}$$

One interpretation of entropy of a message is the amount of choice involved in generating it. Therefore, entropy is closely linked to the probabilities of the various symbols within a message. An extreme is when the content of a message is known by the receiver before it is sent. The amount of choice is zero and so is the entropy. On the other extreme if any symbol in a text of length n is equally probable the entropy will approach $\log(n)$. This result is shown in Equation 3.2.

$$
\begin{aligned}
H &= -\sum_{i}^{n} p_i \log(p_i) \\
H &= -n\left(\frac{1}{n} * \log\left(\frac{1}{n}\right)\right) \\
H &= \log(n)
\end{aligned}
\tag{3.2}
$$

In computer science it is common to extract the *information content* of a single symbol from the entropy definition. The information content is the negative logarithm of the probability of a symbol. The relationship between information content and the entropy is shown in Equation 3.3 and 3.4.

$$I(s) = -\log_2(P(s)) \tag{3.3}$$

$$H = \sum_{s} P(s) * I(s) \tag{3.4}$$

The unit of $I(s)$ is number of bits, for instance a symbol with a probability of 0.25 needs at least $-\log_2(0.25) = 2$ bits to be encoded[WMB99].

3.2 Huffman coding

Huffman codes is an adaptive coding technique. It can be adapted to any probability distribution, and creates an optimal prefix-free code for that distribution. A code is prefix free, if no codeword is a prefix of another. There is no need for special end markers for the decoder to know if the code has ended[WMB99].

Huffman coders builds a Huffman-tree which is a binary tree constructed so that the least probable symbols are placed deep in the tree. The two edges from one node to it is child nodes are given a value 0 or 1. The bit string consisting of the bits encountered in the path from root to a symbol at the leaf is the code for that symbol.

When Huffman codes are decoded, each bit in the input "chooses" one edge in the Huffman tree, when the coder reaches a leaf, the decoder emits the symbol, and starts at the root of the tree.

Symbols placed deep in the tree are assigned longer codewords than those higher up in the tree. This is consistent with Shannon's information theory in that a very probable symbol (with low entropy) should receive short codewords.

Huffman codes are not optimal in the context of Shannon's information theory since it encodes symbols, using a integral number of bits. In a two symbol alphabet

$0, 1$ with probabilities $0.01, 0.99$, 1 should idealy be encoded in less than one bit ($-\log_2(P(1)) = -\log_2 0.99 \approx 0.014$). In order to approach the entropy of such distributions, arithmetic coding is needed. Arithmetic coding is beyond the scope of this report, and it is usually very slow[WMB99].

3.3 Compression of inverted indexes

The literature contains several studies of inverted list compression. In the following sections some of the most popular techniques as well as some new and more novel approaches are presented. Most of these methods are general integer coding techniques and have applications beyond the encoding of inverted files. Several of them were invented in the sixties and the seventies, and the original papers do not even mention inverted indexes as a targeted platform.

Common to all of these methods are that they exploit the fact that even though inverted files might contain large values, the values are generally small. Furthermore, since the inverted files are sorted in a strictly increasing order; it is common that the compression methods works with the gaps between the postings instead of the posting values themselves. The gaps in this increasing sequence contain even smaller values than the original sequence. These assumptions of "small" values give several algorithms where "large" values receive code words longer than in the uncompressed form. The extreme example of this is the *Unary* coding scheme which codes each value as a zero terminated string of one-bits[WMB99].

3.3.1 Bit oriented methods

Several different bit oriented methods have been described, ranging from the extremely simple *Unary* codes [WMB99, ZM06] to the more elaborate interpolative coding[MS00, Tro03]. Other methods are *Elias* γ and δ codes [WMB99, ZM06, Tro03] and the most popular ones are perhaps *Golomb* and *Rice* codes[Gol66, WMB99, Tro03]. Examples of some of the encoding schemes described in this section are shown in Table 3.1.

3.3.1.1 Unary encoding

The unary codes will as described earlier code each value as a string of one-bits terminated by a single zero-bit. The number three will be encoded as **110**. There are not many studies of this encoding scheme; however, it has it uses, especially as part of other bit oriented methods.

In the context of Shannon's information theory (Section 3.1) the unary coding is

x	Unary	Elias-γ	Elias-δ	Golomb	
				b = 3	**b = 8** (Rice)
1	0	0 0	0 0 0	0 0	0 000
2	10	10 0	10 0 0	0 10	0 001
3	110	10 1	10 0 1	0 11	0 010
4	1110	110 0	10 1 0	10 0	0 011
5	11110	110 1	10 1 1	10 10	0 100
6	111110	110 10	10 1 10	10 11	0 101
7	1111110	110 11	10 1 11	110 0	0 110
8	11111110	1110 0	110 0 0	110 10	0 111
9	111111110	1110 1	110 0 1	110 11	10 000
10	1111111110	1110 10	110 0 10	1110 0	10 001

Table 3.1: Examples of Unary, γ, δ and Golomb (Rice) codes.

optimal if the probability a symbol x is as given in Equation 3.5. That is, unary coding is optimal when there are half as many occurrences of $x+1$ than x.

$$P(x) = 2^{-x} \qquad \text{for } x > 0 \tag{3.5}$$

3.3.1.2 Elias γ and δ codes

In Peter Elias's paper from 1975 [Eli75] two coding schemes for integers called γ and δ are introduced. The Elias γ code codes one integer x in two parts, first the number $1 + \lfloor \log_2 x \rfloor$ in unary code, followed by $x - 2^{\lfloor \log_2 x \rfloor}$ coded in binary. The second part of the code only requires $\lfloor \log_2 x \rfloor$ bits. The unary part of the code then gives the length of the binary coded part. Decoding γ codes consists of first extracting the unary part c_u, then read the next c_u bits as a binary value c_b. The decoded integer is then $2^{c_u - 1} + c_b$.

An optimization of these basic Elias-γ codes described in [BC06, Eli75] is based on the insight that if $c_u = k$ then c_b cannot be represented in less than k bits. This again means that the most significant bit in c_b (bit number k) will always be 1, and therefore can be omitted from the codeword. This saves one bit for every codeword and can be implemented very easily. One can choose to "invert" the unary part of the code so that it becomes a run of zeros terminated by a one. Then the single one terminating the unary code might be used as the most significant bit in the binary code, thus, allowing the unary and binary coded codeword overlap. This optimization have not been applied to the compressions shown in Table 3.1.

The encoded length of an integer x is approximately equal to $l_{x,\gamma} = 1 + 2\log_2(x)$ bits. By using the information content definition in Equation 3.4 the probability distribution best encoded by the Elias γ code is given in Equation 3.6.

$$P_\gamma(x) = 2^{-l_{x,\gamma}} \approx 2^{-(1+2\log_2 x)} = 2^{-1}2^{-\log_2(x^2)} = \frac{1}{2x^2} \tag{3.6}$$

An extension over the γ encoding is the δ encoding scheme. This methods uses the γ code to encode the c_u value instead of the unary code. This comes at a price for lower values, however, as the number to be encoded increases the required number of bits decline. The length of Elias-δ codewords, together with the probability distribution best encoded by this method is shown in Equation 3.7.

$$\begin{aligned} l_{x,\delta} &= 1 + 2\lfloor \log_2 \log_2 2x \rfloor + \lfloor \log_2 x \rfloor \\ P_\delta(x) &\approx 2^{1+2\log_2 \log_2 x + \log_2 x} = \frac{1}{2x(\log_2 x)^2} \end{aligned} \tag{3.7}$$

Since $P_\delta(x)$ is larger than $P_\gamma(x)$ for large values of x it is clear that the δ-codes compresses data with high values better than the γ codes.

In [BC06] γ-codes are extended to a "Generalized Unaligned Binary Code" (*GUBC*). Here the length of the binary part of the code is a result of the unary code multiplied with a parameter σ. This means that if the unary code has value of 4 and the σ parameter is 3, the binary part of the code will be 12 bits long. Making it possible to save space when coding lists with large values. The σ parameter can be selected in a brute force search for the best value of $\sigma \in [1, 15]$ in the range, the paper states that this search should not be performed on the entire list to be compressed but the histogram of the values within that list. A even more elaborate scheme proposed is the "Generalized Unaligned Binary Code with n Components" (*GUBC-n*). Here a sequence of values $[\sigma_1, \ldots, \sigma_n]$ is calculated and the length of the binary codeword is calculated as shown in Equation 3.8.

$$|c_b| = \begin{cases} \sum_{i=1}^{c_u} \sigma_i & \text{if } c_u \leq n \\ \sum_{i=1}^{n} \sigma_i + \sum_{i=n}^{c_u} \sigma_n & \text{if } c_u > n \end{cases} \tag{3.8}$$

The motivation behind the *GUBC-n* method is that the probability distributions of the gaps in posting lists might have several different local maximums. At least this is the case for the schema-independent [CCB95a, CCB95b] index used in [BC06]. In the paper a *GUBC-3* method is implemented and is shown to deliver compression ratios comparable to Interpolative coding (Section 3.3.1.4), while query performance is comparable to vByte (Section 3.3.2). It should be noted that a document level index did not show any significant advantage for the new methods in terms of compression ratio.

3.3.1.3 Golomb and Rice encoding

A method introduced in 1966 by Solomon W. Golomb [Gol66] allows for tunable compression. These Golomb codes employ one tunable parameter m and encode a value in two parts, the quotient ($q = \lfloor \frac{x-1}{m} \rfloor$) and the remainder ($r = x - qm - 1$). The quotient is encoded as $q + 1$ in unary code, while the remainder is encoded in binary. The number of bits required to store the binary encoded part either $\lceil \log_2 m \rceil$ or $\lfloor \log_2 m \rfloor$. The extra one bit is required when the remainder is larger than the pivot value $p = 2^{\lfloor \log_2 m \rfloor + 1} - m$. If $r < p$ the remainder is written in $\lfloor \log_2 m \rfloor$ bits, if not $r + p$ is stored in binary [Tro03, WMB99].

In Golomb's original paper the case where m is a power of two is mentioned as a special case where the remainder never requires any more than $\log_2 m$ bits. This means that if one restricts the tuning parameter m to values which are full powers of two encoding and decoding will be easier. Such encodings are called Rice encoding after Robert F. Rice.

The mathematical insight behind the Golomb codes are based on run length coding of a binary event. If one binary event occurs with a probability p. Assuming independence the probability of having one occurrence following $x - 1$ non occurrences follows the geometric distribution shown in Equation 3.9.

$$P(x) = (1 - p)^{x-1} p \tag{3.9}$$

In Golomb's original paper the application is the British secret service agent 00111 playing roulette, communicating his winning run-lengths over a binary channel. However, this model also fits the gaps in an inverted list where the chance of a term occurring in one document is p and x is the gap value.

The choice of the parameter m is very important to the effect of the coding scheme. If large values are to be encoded a small m would lead to high values for the quotient, for small values a high m would lead to a large remainder. In [WMB99] it is suggested that for an array of integers a, a good choice for m is $m \approx 0.69 * \text{mean}(a)$.

The Rice variant of the Golomb codes poses restrictions on the tuning parameter, this leads to some loss in compression ratio, however, these losses are usually surpassed by computationally much simpler and effective implementation.

3.3.1.4 Interpolative coding

Interpolative coding is a method described in [MS96, WMB99, MS00]. Interpolative codes do not work on the gaps in the posting files, but the original strictly increasing sequence. Interpolative codes exploit the fact that if the list $[2, 4, 7, 8, 9]$ is to be

encoded, and assuming that every other value is known, the values between the known pointers have a relative restricted set of valid values. For instance, if 2 and 7 is known and 4 which lies between these two values are to be encoded the only possible values are 3, 4 or 5. Thus, only two bits is required to compress this value. An extreme is when encoding 8 between 7 and 9, zero bits are used.

The list of known values required to encode or decode a sequence is approximately half of those original list. The interpolative coder can be applied to this subsequence as well. Giving a recursive method.

Assuming that a value in a range $1 \dots r$ is to be encoded, using codewords of length $\lceil \log_2 r \rceil$ might be wasting some bits ($2^{\lceil \log_2 r \rceil} - r$ bits to be exact). To counter this the interpolated values can be written in a minimal binary code.

A minimal binary code can be constructed the same way as in the Golomb codes. A pivot is created $p = 2^{\lfloor \log_2 r \rfloor + 1} - r$ values less than p is receive short codewords ($\lfloor \log_2 r \rfloor$) while those larger than p gets the long code word ($\lceil \log_2 r \rceil$). One trick which has shown a slightly better compression ratio is to assign the shorter codewords to the center of the range [MS00].

Interpolative coding has been shown to achieve remarkable good compression ratios [MS96, Tro03, MS00, AM05, BC06, AM04], this is due to the ability to exploit local clustering in the inverted files. However, the recursive and somewhat complex nature of the algorithm makes its implementations slow. However, the results in studies like [Tro03] finding interpolative compression approximately ten times slower per decoded value than for instance *vByte* (see Section 3.3.2) might be to pessimistic. In [BC06] a highly optimized version of the interpolative algorithm performed is only three or four times slower than *vByte*. This suggests that there are much performance to be gained by implementing the bit-oriented algorithms correctly.

3.3.1.5 Selectors

An idea used in the word-aligned compression schemes (Section 3.3.3) is to let a part of the encoded data specify the number of bits for a selected number of values. This idea is applied in [AM04] to create the *Selectors* algorithm.

The *Selectors* method introduces two concepts, *width* which is the bit-width for the next values, *span* which is the number of codes to be encoded with the selected width. Selectors emitted in the encoded data govern the transitions between various widths and spans. These transitions are made relative to the previous encoded chunk and a transition table shown in Figure 3.1. The transition table should of course be adjusted when the width of the previous encoded chunk is near the end of the ranges so that no selection codes are wasted.

Width	Span		
	s_1	s_2	s_3
-3	Code 1		
-2	Code 2	Code 3	
-1	Code 4	Code 5	Code 6
0	Code 7	Code 8	Code 9
$+1$	Code 10	Code 11	Code 12
$+2$	Code 13	Code 14	
$+3$	Code 15		
max	Code 16		

Figure 3.1: Selectors transitions as described in [AM04]

The assignment of selectors, might be seen as a shortest path problem. The nodes in the graph are pairs (p, w) where p is the position in the list to be compressed, and w is the width used previously. Edges are placed in the graph for every selector code which "fits", for instance a code 9 edge from (p, w) to $(p + s_3, w)$ is only possible if the s_3 elements after p need no more than w bits. The cost of one edge is $h + sw$ where h is the size of the selector, s is the span and w is the bit width used.

An artificial node is created $(0, w_{max})$ and used as the start of a one-to-all shortest path. There are at most w_{max} nodes at the nth position and the one with the lowest cost represents the path which should be used to encode the list. The shortest path algorithm for directed acyclic graphs (DAGs) is linear ($O(n)$) where n is the number of nodes in the graph[CLRS01], however, it still comes at a cost.

A greedy method is suggested, one traverses the graph by choosing the "best" edge out of each state. The definition of the "best" edge used in the paper is the edge with the lowest cost defined as: the savings due to high spans minus the bits wasted due to to high bit-widths. This much simpler method gives very good results.

Homogeneous or structured lists where large portions of the lists contain similar values benefit from high span values amortizing the cost of the selector over more values. Several different choices for the span values $[s_1, s_2, s_3]$ are experimented with, and the overall best compression ratios are achieved with $[2, 4, 6]$ or $[2, 4, 8]$.

Further improvements are achieved when a *multiplier* parameter is selected for each list. This multiplier adjusts the span values so that homogeneous and structured lists receive large span values, while more noisy lists are encoded using shorter spans. The authors of the paper suggest a brute force search for the optimal multiplier parameter.

Another trick used to encode homogeneous lists more efficiently is the insertion

of a 4-bit *escape* after each sequence encoded using the s_3 span. This escape represents an extension of the encoded run with a number of values in the range $[0, 15 * \text{multiplier}]$.

Both the *multiplier* and the *escape*-nibble give improved compression ratio for the test collections used the paper. But the compression rate is still outperformed by Golomb and Interpolative codes. Retrieval time is improved over both *vByte* (Section 3.3.2) and *Carryover-12* (Section 3.3.3.3).

3.3.1.6 PFOR and PFORDelta

Introduced in [ZHNB06], the PFOR and PFORDelta methods are motivated by current hardware architecture. Modern CPUs are said to be super-scalar, that is; they are able to issue multiple instructions per clock cycle. However, to exploit this ability fully, the algorithms running should exhibit certain characteristics. The focus when designing the PFOR and PFORDelta methods were the following:

- **The data should reside in *cache-memory*:** Avoiding the extra cost of fetching data from main memory on cache misses.
- **Branches (`if-then-else`) should be avoided:** This avoids miss predicted branches causing CPU pipelines to be flushed and several cycles lost.
- **Iteration in loops should be independent:** This allows compilers and CPUs to *vectorize* the program, performing instructions on several pieces of data in one clock cycle.

PFOR is short for *Patched Frame of Reference*. It is an optimized version of a method called *Frame of Reference* (FOR). In FOR one maintain a single value min_c for each disk block, and every value $c[i]$ is encoded as $c[i] - min_c$ using $\lceil \log_2(max_c - min_c - 1) \rceil$ bits where min_c is the smallest value in the block, and max_c is the largest. PFORDelta is an extension of PFOR useful for sorted lists. In PFORDelta the gaps in the list of numbers are stored, rather than the numbers themselves, this makes PFORDelta suitable for inverted list compression. It seems that in litterature the name PFORDelta are often used even when deltas are not encoded, for instance [YDS09b] uses a PFORDelta method to compress term frequencies, which will probably not encode the gaps, since the gaps of frequencies can contain many negative numbers.

Since FOR uses $\lceil \log_2(max_c - min_c - 1) \rceil$ bits for each value; outliers could damage the compression ratio significantly. Therefore PFOR selects a code width smaller than than this. Values which require higher values are coded as exceptions. This allows for good compression when there are only few large values.

The major contribution of the PFOR methods are the *Patching* mechanism. The main body of the data is encoded in the "code section" all using b bits per value.

For values that are exception a relative pointer to the next exception is stored. This way, by storing a pointer to the first exception in the preamble of a encoded block it's possible to traverse the linked list of exceptions. Furthermore it poses a requirement that the distance between exceptions must be less than $2^b - 1$. If this is not the case a value that otherwise would fit in b bits is coded as a compulsory exception. An example of a list of numbers compressed using PFOR is shown in Figure 3.2. In the first phase of the algorithm all the data in the "code section"

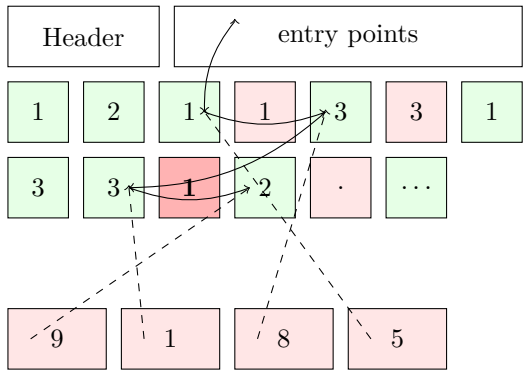

Figure 3.2: A PFOR compressed block for the sequence $[1, 2, 1, 5, 3, 8, 1, 3, 3, 1, 2, 9]$. One forced exception is showed with bold typesetting. Exception regions grows backwards.

is unpacked. Then the linked list of exceptions are followed and the exceptions patched into the list. The PFOR algorithm is applied to chunks of values, the paper suggests that a chunk size of 128 is suitable. Highly optimized code can be written which packs and unpacks a chunk of 128 b-bit values. Such a loop can be written without branches, with independent iterations. With a chunk size of 128 the data will fit in cache on most modern architectures. A nice property obtained since one select chunk sizes which is a divisible with 32 is that independent of the selected b-value; the compressed data will be aligned on a 32-bit machine word boundary.

The patching phase does not follow the property that every iteration in the loop should be independent, however, one should choose b in such a way that the number of exceptions are small.

3.3.1.7 NewPFD and OptPFD

The compulsory exceptions of PFor and PForDelta become more frequent as one chooses low b values. For instance, if one chooses a b value of 2, every exception cannot be more than 4 spaces apart. To alleviate this limitation, [YDS09b] intro-

duces two new methods similar to PForDelta. Theese two methods are *NewPFD* and *OptPFD*. When encoding a block with a given b value, one stores the lower b bits of any exception inside the encoded block. The remaining *owerflow* bits, and the position of the exception are stored in two separate arrays, which in turn are compressed using another method. In [YDS09b] the overflow values and positions are compressed using the Simple-16 method. In another paper by the same authors [DHYS08] the NewPFD algorithms were applied recursively on the exceptions and exception positions.

In the original PForDelta paper the b value were selected so that close to 90% of the values where within range. In [YDS09b] the authors discuss that a fixed threshold such as 90% is not optimal. Assuming that the decompression speed if a PFor compressed block is goverened by the number of exceptions, one can choose to increase the b value, thus lowering the number of exceptions, but possibley increasing the compressed size in order to trade space for speed. The paper states that all blocks are initially compressed using the b value which yields the smalled index size. Then one might increase the b value for the block which will give the most time savings per increase in space.

The experiments performed in [YDS09b] use a document collection which is sorted to yield smaller gaps in the index. That is, the documents were ordered so that documents sharing many terms recieved document identifiers which were close to each other. For this collection, the NewPFD and OptPFD methods were significantly better than the original PForDelta method.

3.3.2 Variable byte length coding (*vByte*)

One very popular method described [WZ99] is the variable byte length encoding (*vByte*). The method is extremely simple and can be implemented efficiently. Each code word is a whole number of bytes, this leads to fast decoding since most computers can handle operations on single bytes faster than operations on single bits. The simplicity of the algorithm itself also contributes to its high-throughput nature.

vByte works by using seven bits per byte to code a value and one bit per byte to signal if the that byte is the last byte in the codeword, or if there are more bytes. The operation to decode one byte is very simple. Assuming that the signal bit is the most significant bit in the byte on can use the `AND` operation and the bit-mask 01111111 to isolate the value, then a accumulator variable might be shifted seven bits to the left and the masked byte can be added to the accumulator. This operation is shown in Equation 3.10.

$$v = (v << 7) + (b \ \& \ 01111111) \tag{3.10}$$

vByte does not offer especially good compression ratios. For instance very small gaps which might be encoded in zero or just a couple of bits using a interpolative coder will consumes a whole byte using vByte. The pseudocode for the vByte encode and decode operation is shown in Algorithm 1 and 2.

Algorithm 1: vByte encoding

```
   Input: val
   Output: byte stream
1  count ← 0
2  while val ≥ 128 do
3  |   WriteByte((val & 127) | 128)
4  |   val ← val >> 7
5  |   count ← count + 1
6  end
7  WriteByte(val)
8  return count
```

Algorithm 2: vByte decoding

```
   Input: byte stream
   Output: decoded number
1  value ← 0
2  repeat
3  |   byte ← NextByte()
4  |   value ← value << 7
5  |   value ← value + (byte & 127)
6  until byte < 128
7  return value
```

There exists other byte aligned codes which support some parameterization. These codes are called (S, C)-Dense codes and are described in [BFNE03]. The usage of (S, C)-dense codes are not that widespread as vByte, probably because the fast bit shifting and masks in the vByte algorithm has to be replaced with slow operations such as division and multiplication in (S, C)-Dense.

3.3.3 Word aligned codes

Recent publications [AM05, AM06] have shown algorithms which show decompression speed comparable with *vByte* while acheaving much better compression ratios. While the compression ratios of these methods are worse that those of the best bit-oriented methods such as *Golomb* (Section 3.3.1.3) and *Interpolative* coding (Section 3.3.1.4) decoding speeds are much higher.

3.3.3.1 Simple-9 and Simple-16

Simple-9 was introduced in [AM05] and works on whole 32-bit long machine words. Of these words four bits (the selector) are used to select a scheme for how the remaining 28 bits are to be used. Simple-9 has nine such schemes. These schemes are shown in Table 3.2.

Some of the various schemes do not utilize all the bits the machine word, and only nine of the sixteen distinguishable schemes a four bit selector can address is used. These observations have led to the *Simple-16* scheme described in [ZLS08]; for

Selector	Values	Length	Wasted bits
a	28	1	0
b	14	2	0
c	9	3	1
d	7	4	0
e	5	5	3
f	4	7	0
g	3	9	1
h	2	14	0
i	1	28	0

Table 3.2: Simple-9 coding schemes as described in [AM05].

instance the case for the selector **e** in Table 3.2 wastes three bits, but in Simple-16 it is replaced by two different schemes; one with three six-bit values followed by two five-bit values, and another with two five-bit values followed by three six-bit values. [ZLS08] does not provide a listing of the cases. In [YDS09b] a version of Simple-16 called Simple-16-128 is described, this version allocates most of the selectors for small values and fewer for the large ones.

[AM05] shows that Simple-9 performs on par with *vByte* in terms of decompression speed while it provides compression ratios better than *vByte*. Similar results are shown in [YDS09b, ZLS08] for all different Simple-9/16 versions.

3.3.3.2 Relative-10

Introduced in the same paper as Simple-9 ([AM05]) Relative-10 makes better use of the selector bits by choosing one using the scheme employed in the previous encoded block when coding the next block. Each block is encoded relative to the previous one. Relative-10 only uses two bits in each machine word to select the coding scheme and has 30 bits left to encode the actual value. The various coding schemes described in the original paper is shown in Table 3.3.

Relative-10 works well if the inverted list is homogeneous, minimizing the impact of the limited ability to choose the selector values. Since Relative-10 permits 30 data bits; it performs very well on small values, packing more one and two-bit values into each code word. Relative-10 achieve compression ratios slightly better than Simple-9 with equal decompression speed.

If one knows the maximal value within the list to be compressed, the relative scheme selector table can be shifted to the left. That is, the maximal code "escape" (selector 3 in Table 3.3) can escape to a lower value, thus yielding better compression ration. In the experiments performed in the original paper [AM05], the maximal encoding scheme required for a given block where encoded in a header field of the

Selector	Count	Length	Wasted	Next selector									
				a	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**
a	30	1	0	0	1	2							3
b	15	2	0	0	1	2							3
c	10	3	0		0	1	2						3
d	7	4	2			0	1	2					3
e	6	5	0				0	1	2				3
f	5	6	0					0	1	2			3
g	4	7	2						0	1	2		3
h	3	10	0							0	1	2	3
i	2	15	0							0	1	2	3
j	1	30	0							0	1	2	3

Table 3.3: Relative-10 coding schemes as described in [AM05]

block, enabling this optimization.

3.3.3.3 Carryover-12

Clearly Relative-10 provides some improvements over Simple-9, however, as shown in Table 3.3 there are still selectors which waste bits. Measures similar to those taken in the Simple-16 variants could probably be employed, but [AM05] describes another method called *Carryover-12*. Instead of creating additional cases for the ones where bits are wasted, these wasted bits are used to store the selector for the next codeword. This way the next codeword can employ all of its 32 bits to store data.

This results in two different encoding schemes, one scheme where the previous codeword contained the selector of the current word, and another scheme when the word itself need to contain the selector. The two schemes is shown in Table 3.4. The transitions between various selectors are implemented in the same way as in Relative-10.

The compression ratios of Carryover-12 is better than those of Relative-10 and Simple-9, however its slightly more complex implementation makes the decompression time higher. However, the decoding speed is still comparative with *vByte* and [AM06] also reports this same findings.

Carryover-12 can take advantage of the same optimization with respect to the max value escape selector as the Relative-10 method.

Selector	Previous selector			No previous		
	Count	**Length**	**Next**	**Count**	**Length**	**Next**
a	32	1		30	1	
b	16	2		15	3	
c	10	3	Yes	10	3	
d	8	4		7	4	Yes
e	6	5	Yes	6	5	
f	5	6	Yes	5	6	
g	4	7	Yes	4	7	Yes
h	4	8		3	9	Yes
i	3	10	Yes	3	10	
j	2	15	Yes	2	14	Yes
k	2	16		2	15	
l	1	28	Yes	1	28	Yes

Table 3.4: Carryover-12 coding scheme as described in [AM05]

3.3.3.4 Slide

Slide is a word aligned scheme presented in [AM06]. *Slide* works somewhat similar to Carryover-12 packing minimum width binary codes into machine words. However, Slide makes full use of the trailing bits at the end of a partially full codeword.

Slide permits codes of any length, unused bits at the end of one codeword is prefixed at the beginning of the next word. Since Carryover-12 does not allow this kind of word-boundary spanning there is no selector for 13-bit codes, it would waste more, and it would not allow any more values than if 15-bit words were used. The availability of all different codeword widths removes this internal fragmentation. Also since unused space at the end of a encoded value s prefixed the next word external fragmentation is also reduced.

In Slide, selectors are three bits wide. These describe a relative shift of the codeword length just as in Relative-10 and Carryover-12, one selector values is reserved for "Use the widest codewords available" to accommodate sudden jumps in the values. Given a previous code word length s then the selector might choose values in the range $[s-3,\ldots,s+3,s_{\max}]$. The three bit selectors is not really enforced by the algorithm, the paper mentions that the impact of choosing either two-bit selectors (three relative values) or four-bit selectors (fifteen relative values) is quite small.

A scheme for handling zero-bit code words are also sketched out. The selector might set the code width to zero, and then a fixed width binary number might state the number of zeros in the run. But this adds complexity to the algorithm which might result in to slow decompression, so representing zeros as one-bit codes might be just as good a solution.

The authors of [AM06] argue whether this encoding scheme still is word aligned

or not. The codes span word boundaries and require somewhat more complex decoding mechanisms, but the data which is accessed is fixed width binary codes, and no operations are performed on single bits. In experiments the Slide algorithm performs better than Carryover-12 in terms of compression ratio, and comparable with both *vByte* and Carryover-12 in decompression speed.

3.3.4 A taxonomy of inverted list compression techniques

The make it easier to discuss the methods used to compress inverted lists the various methods could be divided in several classes. In this section a simple taxonomy for the various methods will be described. The taxonomy will contain three axes, these are summarized below. The methods described in this chapter are classified using the taxonomy in Table 3.5.

	Single value			**Chunk of values**		
	Bit	**Byte**	**Word**	**Bit**	**Byte**	**Word**
Parameterized	Golomb Rice GUBC GUBC-n Huffman	(S,D)-dense		Selectors PFOR PFORDELTA NewPFD OptPFD		
Non parameterized	Elias-γ Elias-δ Unary	vByte		Interpolative		Simple-9 Simple-16 Relative-10 Carryover-12 Slide

Table 3.5: Taxonomy of compression methods.

3.3.4.1 Individual value or chunk

When implementing inverted list compression special care has to be taken if the data is to be compressed using a method which works on multiple values at the same time. Method like *vByte* and the *Elias* methods can be used on single values as well as long runs of values. However, other methods like Simple-9, Carryover-12,

Interpolative and PFOR require multiple values to be encoded at once. The layout of the inverted file need to be adapted if such methods are to be used.

3.3.4.2 Bit, Byte and Word alignment

A classification already employed in this chapter is the level of alignment offered by the various algorithms. The alignment of encodings affects the decompression performance, but it also affects how various encodings might be used together. For instance a bit oriented encoding such as the *Unary* code would be wasteful together with the byte aligned *vByte* method since one would have to pad the unary codewords to get byte aligned.

3.3.4.3 Parametrized vs. non parametrized

Some methods are parameterized, one or more parameters are chosen to adapt the compression method to the data. This parameter might be chosen globally or for some local quantity. The latter will often yield better compression ratio, but at the cost of some local analysis of the data. Examples of parameterized codes are Golomb codes (the m parameter) and PFOR (the b parameter). Examples of non parameterized codes are vByte, unary codes and the Elias codes. This categorization is also used in the Managing Gigabytes book [WMB99].

3.3.4.4 Using the taxonomy

The taxonomy described above might be used as a guideline when selecting compression methods. For instance, if one has an inverted list layout which is something like $(d, f_d, p_1 \dots p_{f_d})^+$ compressing document ids and frequencies using a chunk method will be hard. As described in [AM05] chunk methods are vulnerable to value distributions like $[1, l, 1, l \dots]$ where l is some relatively large value. All known chunk methods will encode the low values (the ones) using to large codewords. This suggests that even though the inverted list might be looked upon as just a sequence of integer, chunked compression will benefit from compressing "similar data" together. Thus, a layout resembling a column store like the one shown below will be easier to adapt to a chunk method.

$$(d^+, f^+, (p_{d_1,1} \dots p_{d_1,f_1}) \dots (p_{d_k,1} \dots p_{d_k,f_k}))$$

That said, chunk methods which do not require very long chunks (for instance Simple-9) could be used compressing positions in former list layout. This leads to another possible problem, most "short-chunk" methods are word aligned, this

means that one might pad the representation of the document id and the frequency to be able to start the position list at a word boundary.

Efficient query evaluation requires other access methods than just sequential scan through the posting files [MRS09d]. This often involve skipping through the list reading only some values. Parameterized codes would require special attention when skipping since one jump might hit a section of the posting which is compressed with another "parameter" than the other. This also applies for alignment, codes with a larger unit of alignment will often be easier to skip into. Chunk methods will be difficult to skip into, especially the relative coding schemes such as Carryover-12, Slide and Relative-10 where the coding of one chunk depends on the coding of the previous one.

None of these problems are impossible to bypass, skip pointers might include the "context" required to start decompressing the data at the target site, or perhaps reset the "context" at every skip pointer position. However, this will make random access structures more expensive. And therefore this should be taken into consideration when choosing compression method.

Index construction will often include merging already created sub indexes[WMB99]. Assuming that one sub index contains postings for documents with identifiers in the range $[d_0 \dots d_k]$ and the other sub index contains the posting for documents in the range $[d_{k+1} \dots d_n]$ merging these the lists for each term in these indexes could be done by just concatenating them. This is trivial to achieve when using unparameterized single value methods, and parameterized single methods where the parameters are equal. However chunk method, and parameterized methods with different parameters might require the two postings to be uncompressed before they are concatenated and compressed again.

The last concern is the memory requirements. For parameterized codes one will require some amount of the data to reside in memory in order to estimate the best values for the parameters. This is not the case for the unparameterized codes where the data can be outputted "on-the-fly". Also chunked methods will require some additional memory, to collect the values before writing them in chunks.

3.3.4.5 Taxonomy conclusion

Since most of the classes except perhaps the single value non-parameterized comes with some limitations or will require some extra care. Taking into account that these methods are among the simplest to implement this might explain their widespread usage.

If one is to choose a method from the other classes it would be to bypass the less-than-optimal compression ratio and compression throughput offered by the non-parameterized single value codes. When building an inverted index system from scratch more options will be available. One should for instance consider a

column-store layout, which will make it possible to utilize the full potential of the chunk methods.

3.4 Compression of dewey codes

Dewey codes will in its uncompressed form occupy much storage. Compression should give improvements on this, and in this section recent work on dewey compression will be described. The dewey node labeling scheme assumed in this report is only one of many, and several approaches exists, however, an exhaustive study of these methods is beyond the scope of this report. There exists a lot of research on labeling schemes for XML documents. [SCCS09] is an up-to-date survey of the various schemes available.

3.4.1 Node labeling schemes for dynamic XML documents

The focus in [HHMW07] is a Dewey labeling scheme which allows modifications to the documents. A labeling scheme which allows efficient insertion of new nodes in existing XML documents poses requirements far beyond the simple Dewey codes assumed in this report. However, the paper describes several possible compact methods to represent Dewey codes, several of which can be applied in this study.

3.4.1.1 Desirable properties of dewey encoding

Even though any general integer coding technique such as those described in the previous sections can be used to encode deweys, some methods will be better than others. [HHMW07] describes some extra features which are desirable when compressing deweys.

- **Order preserving:** If two Dewey codes D_1 and D_2 subject to $D_1 < D_2$ then the compressed versions should also be subject to $C(D_1) < C(D_2)$. This makes it possible to answer "occurs before" and "occurs after" predicates without decompressing the deweys.
- **Prefix comparable:** If for instance two Dewey codes are equal on their first three elements, then the compressed versions should be equal on some prefix corresponding to the first three elements as well. This means that when searching for prefix-equality one might just compress the search key and compare the compressed data to the compressed key.

When using compression methods which have these properties, query-processing can be faster than when working with uncompressed data. For instance, if one were

to compare two uncompressed Deweys one might need to execute one operation per dewey element, if the data was compressed so that several elements would be packed for instance inside one machine word multiple elements might be compared in one operation.

Some of the compression method described in Section 3.3 have these features or can be adapted with little effort. For instance vByte (Section 3.3.2) has these feature when the "signal"-bit is the most significant bit in the code. All the methods described in Section 3.3 which is order preserving and prefix comparable are shown in Table 3.6.

Method	**Remarks**
vByte	Most significant bit as "signal"-bit and Big-Endian byte order.
Golob and Rice	Given that the search key are compressed with the same b-parameter value.
Unary	
Elias-γ and δ	

Table 3.6: Order preserving and Prefix comparable codes

3.4.1.2 Dewey coding techniques in [HHMW07]

In the following the techniques described in [HHMW07] will be adapted to the simpler static use-case of dewey codes which is the context of this report. The number of bits required to code the different dewey element values using the mentioned methods are shown in Table 3.7.

Static scheme, with prior knowledge If one knows the largest values within each level in the Dewey codes a fixed bit-width might be assigned to each level in the Dewey codes. If the values within one level is very skewed this might cause this compression to be very inefficient.

Given that the prior knowledge of the maximal values within each level this encoding can probably be implemented quite efficiently. However, to obtain good compression ratios one should probably encode relatively small chunks of Dewey codes using local maximums to determine bit-widths, making the impact of sporadic large values smaller. On the downside this will break order preservation and prefix comparison. If one choose to encode the Dewey codes using the global maximal values the order preservation and prefix comparison properties will be kept, but the achieved compression ratio could be very low.

Encoding methods		**Code length for value**				
		7	2^7	2^{14}	2^{21}	2^{28}
Optimal		3	7	14	21	28
Length fields						
Fixed length	4 bits	7	11	18	-	-
	5 bits	8	12	19	26	33
Stepwise growth	$u = 2$	5	13	24	34	48
	$u = 3$	6	**10**	20	30	40
	$u = 4$	7	11	18	29	36
Two part	2 bits	7	12	19	27	34
	3 bits	8	13	20	28	35
Control tokens	$k = 3$	6	12	21	33	42
	$k = 4$	**4**	12	20	28	40
	$k = 5$	5	**10**	20	30	35
Separators	$m = 2$	6	12	20	30	33
	$m = 3$	9	12	18	27	33
Prefix free						
	Bit aligned	**4**	12	21	29	36
	Byte aligned	8	16	**16**	**24**	**32**

Table 3.7: Bits required for dewey elements using methods from [HHMW07]. The best methods are shown in boldface.

Fixed width length field: By prefixing every element in the dewey with a fixed width field specifying the bit-width of the next element. Such a variable length encoding can handle sporadic large values without wasting to much space for the smaller values. However, the length of this fixed width field need to be determined. The cost of this length is paid for every dewey element, and a too large value for this field will waste storage.

The authors introduce a technique which is applied to all the next methods described in the article. This is a well known method called *range expansion*. Range expansion makes use of the information that if a value v requires n bits, then v is no smaller than 2^n, thus when encoding v one can write the value $v - 2^n$. This makes the range of possible values being encoded in a variable width codeword somewhat larger.

If the length of the code is L_f and the element to be encoded e_i then Equation 3.11 must hold.

$$e_i \leq \sum_{j=1}^{L_f} 2^j = 2^{L_f+1} - 2 \rightarrow L_f = \log_2(e_i + 2) - 1 \tag{3.11}$$

One could use the methods from the static scheme where the maximal values decided the code word width to decide the width of the length field for each level in the dewey. However, a non-global configuration of the length field would break the order preservation and prefix comparability properties of the encoded values.

This method is far from optimal in that it wastes bits for the length field when encoding small values. This could be countered by letter the length of the length field be variable.

Variable length length field: The paper describes two schemes for variable length length fields. Both schemes are both prefix comparable and order preserving. The first version uses a stepwise growing length field. The length field consists of code-units of length u, one of the 2^2 possible codewords is used to signal that there will be one additional codeword. In Table 3.8 shows codeword lengths when $u = 3$ and 111 is the continuation codeword.

Length field	Codeword length	Length field	Codeword length
000	1	110	7
111 000	8	111 110	14
111 111 000	15	111 111 110	20

Table 3.8: Stepwise code length growth.

It is clear that this length field will be very long for the larger values. The second scheme counters this by creating a two part length descriptor. The first part has fixed length and will describe the length of the second one. By using range expansion and by letting the length of the first field be two bits, code lengths shown in Table 3.9.

Length field	Codeword length	Length field	Codeword length
00 0	1	00 1	2
01 00	3	01 11	6
10 000	7	10 111	14
11 0000	15	11 1111	30

Table 3.9: Two part length descriptor

This encoding consumes fewer bits for the length descriptor, and is therefore superior to the first version. However, if one need to encode values larger than $2^{31} - 1$ two bits for the fixed length selector is not enough. Adding an extra bit to that field will in practice reserve quite a lot of wasted coding lengths.

Both the variable length selector schemes add extra cost for really small values. Thus, they are not optimal, however, especially the two-part scheme seems practical

with respect to implementation, and will provide reasonable compression.

Control tokens: The paper then describes one approach very similar to vByte (Section 3.3.2). A special prefix is added to each encoded element, this prefix is in unary code and represents the number of "encoding-units" used for the element. Each encoding-unit has a fixed size k.

Range expansion is applied so that when encoding a value which requires m encoding units one subtracts the maximal value which can be encoded in $m-1$ units. When $k = 8$ this scheme will produce a byte aligned encoding.

The relationship to *vByte* is that this if the unary prefix of the encoded value is spread across all the encoded units as "signal bits" this will become a *vByte* encoding. This control token method is actually a possible improvement of the standard *vByte* algorithm, allowing the first byte to carry all the signal bits which will allow decoders to just look at this first byte when determining the codeword length. The simple *vByte* method would benefit form range expansion as well.

Examples of the variable length codes with $k = 4$ is shown in Table 3.10. This code will provide nibble-alignment, which in turn will make entire deweys compressed using this method nibble aligned. This nibble alignment means that at most four bits are wasted if the code need to be byte aligned for some underlying data structure (which in fact is the case for the XML database system described in the paper).

m	**Pattern**	**Value range**
1	$0xxx$	0 - 7
2	$10xx\ xxxx$	8 - 71
3	$110x\ xxxx\ xxxx$	72 - 583

Table 3.10: Example of control token codings with $k = 4$

Separators: Another approach using encoding units is the separator method. The encoded dewey consists of several m-bit units. The units are interpreted as digits in a Base-$(2^m - 1)$ number system. One of the possible encoding units is reserved as the level separator. If $m = 2$ and the digits assigned to each codeword is $00 = 0$, $01 = 1$, $10 = 2$ and $11 = .$. The dewey 1.3.7 is shown below.

01 11	01 00 11	10 01
$(1 * 3^0)$	$(1 * 3^1) + (0 * 3^0)$	$(2 * 3^1) + (1 * 3^0)$
1.	3.	7

Making the separator an explicit code unit might not be optimal when compressing

really small element values. Also this method will break comparability and it is not order preserving, since digits and separators might be compared to each other.

Prefix free codes: The last proposed method is to use a prefix free variable length code to specify the length of the encoding for each element in the Dewey. This prefix free code can be determined using a Huffman algorithm (Section 3.2). By letting Huffman-codewords be assigned in increasing order for increasing values the code is prefix comparable and order preserving.

The dewey elements are encoded using the Huffman derived prefix free code C_i and the encoded element with length L_i.

Prefix coding: The most efficient scheme described in this paper is the use of prefix coding which is applied for each storage block. When storing deweys the first dewey, the reference dewey, is stored in its uncompressed from. Subsequent deweys are coded in two parts. The first part is the length of the common prefix. The second part is the uncompressed representation of the remaining elements.

It is possible to encode the remainder using one of the other compression methods. This case will improve compression even more, but it will also introduce one interesting property. Since the prefix compression cuts away the dewey elements close to the root, very many of the lowest dewey element values are removed. Thus, the average value of each dewey element increases which will impact the choice of coding method for the remainder part.

The paper provides a coding scheme which is optimized for the remainder of the deweys when encoded using the prefix coding. Also the scheme are adapted to produce byte aligned codes, wasting no bits on padding and supporting efficient byte level comparisons.

Prefix coded deweys are not directly prefix comparable and order preserving, however, when comparing one "query"-dewey one might encode that dewey relative to the same reference dewey and using the same compression for the remainder. This representation retains the prefix comparable and order preserving properties.

3.4.1.3 Paper conclusion and relevancy for this project

The paper reports that prefix coding alone yields improvements better than 40%. Together with the Huffman based prefix free codes the compressed sizes close to 20-30% of the uncompressed size. Taking into account that the dewey scheme used in the paper assigns higher element values in order to support updates. The compression result might be even better for static dewey codes.

The prefix compression scheme described might not lend it self equally well to

inverted lists since the deweys in each posting are from many different documents. However, the idea of not repeating prefixes will surely be useful. A scheme where the reference dewey are chosen more frequently might provide better compression ratios for inverted lists, but at the cost of more complex query processing.

Many of the methods for encoding the dewey values are not very different form many of the inverted list compression methods shown in Table 3.6. But they point out some possible improvements such as the use of *range expansion* to improve the compression ratio of for instance *vByte*.

3.5 Skipping

Calculating the intersection of two or more inverted lists is a very important operation in may IR applications. These lists are typically sorted on the document identifiers. The problem is therefore to merge the lists as fast as possible. Skipping can speed up this merging process[MRS09d].

One approach is to divide the index into blocks of n entries. Each such block is annotated with a header containing the first document identifier in that block, and a pointer to the beginning of the next block. During query processing the header of the first block is decoded and the pointer to the next block is used to decode the next block. Then one can determine whether the target document identifier is inside the first block ($d_{block_1} \leq d_{query} < d_{block_2}$).

$$\langle 5,1\rangle \langle 8,1\rangle \langle 12,2\rangle \langle 13,3\rangle \langle 15,1\rangle \langle 18,1\rangle \langle 23,2\rangle \langle 28,1\rangle \langle 29,1\rangle \dots$$

(a) Original inverted index (d_i, f_i) pairs.

$$\langle 5,1\rangle \langle 3,1\rangle \langle 4,2\rangle \langle 1,3\rangle \langle 2,1\rangle \langle 3,1\rangle \langle 5,2\rangle \langle 5,1\rangle \langle 1,1\rangle \dots$$

(b) d-gaps (d_i, f_i) pairs.

$$\langle\langle \mathbf{5}, \mathbf{a_2}\rangle\rangle \langle 5,1\rangle \langle 3,1\rangle \langle 4,2\rangle \langle\langle \mathbf{13}, \mathbf{a_3}\rangle\rangle \langle 1,3\rangle \langle 2,1\rangle \langle 3,1\rangle \langle\langle \mathbf{23}, \mathbf{a_4}\rangle\rangle \langle 5,2\rangle \langle 5,1\rangle \langle 1,1\rangle \dots$$

(c) Skip pointers with a block size of three. a_i is the address of the begining of skip block i.

$$\langle\langle \mathbf{5}, \mathbf{a_2}\rangle\rangle \langle 1\rangle \langle 3,1\rangle \langle 4,2\rangle \langle\langle \mathbf{8}, \mathbf{a_3} - \mathbf{a_2}\rangle\rangle \langle 3\rangle \langle 2,1\rangle \langle 3,1\rangle \langle\langle \mathbf{10}, \mathbf{a_4} - \mathbf{a_3}\rangle\rangle \langle 2\rangle \langle 5,1\rangle \langle 1,1\rangle \dots$$

(d) Skip pointers gap encoding and document id of first entry removed from the list.

Figure 3.3: Skip pointers as proposed by Zobel and Moffat in [MZ96].

In Figure 3.3 the structure of skip pointers as described in [MZ96] are shown. In 3.3c skip pointers with a block size of three are added to the index. In Figure 3.3d some redundancy are removed, and the document identifiers of the skip pointers are gap coded. This data structure will allow skipping to traverse the inverted index in larger blocks, this way a smaller portion of the index will need to be decoded.

The effect of skipping is as mentioned above that only parts of the index will need to be read. However, the granularity of the skips affects efficiency. A fine grained skip list with small blocks will make it possible to skip several times, however, it would cause some overhead related to the comparison with skip pointers needed. Skip pointers also require storage space. A coarser skip list will result in fewer skips, but also fewer comparison and less storage. In [MRS09d] a simple heuristic for skip list granularity is suggested. For a posting list of length P, it is suggested to use $\sqrt{P}$ evenly spaced skip pointers.

In [MZ96] a more elaborate model is proposed. This model assumes that there are p postings in the inverted list. Among these p postings, there are k documents of interest. Assuming that each of the k documents are within separate blocks one would on average need to decode half of each such block to find the document. If there are p_1 skip pointers (and blocks) each block will contain $\frac{p}{p_1}$ postings. Additionally one would have to decode the skip pointers as well, the size of the skip pointers is approximated to the size of two regular postings ((d_i, f_i) pairs). This gives the total processing time shown in Equation 3.12. In Equation 3.12 t_d is the cost of decoding one posting and T_d is the total decoding time.

$$T_d = t_d \left(2p_1 + \frac{kp}{2p_1} \right) \tag{3.12}$$

The decoding time is minimized when $p_1 = \frac{\sqrt{kp}}{2}$. These numbers do not take the extra IO requirements from the skip information. Assuming that the cost of reading one posting is t_r the total cost is given in Equation 3.13.

$$T = t_d \left(2p_1 + \frac{kp}{2p_1} \right) + t_r(p + 2p_1) \tag{3.13}$$

The total time is minimized when $p_1 = \frac{\sqrt{kp/(1+t_r/t_d)}}{2}$. The paper also discusses the use of several levels of skips. Skipping in the skip pointers reduces the processing required, however, such pointers require extra storage. In [MZ96] equations describing the costs of multi level skip lists show a slight advantage for skip lists of level two and three over the single level approach. However, in a real world implementation these improvements could be offset by a more complex implementation.

The k parameter need to be known when the index is built. In [MZ96] experiments are performed with different k values. High k values is favored by boolean queries with few terms, while longer queries favor low k values. The paper concludes that a k between 100 and 1000 is appropriate for most queries. Assigning different k values to different lists are suggested as one optimization. However, it's not simple to determine the best k values for any given list.

A slightly different approach for in-memory indexes are used in [SC07]. Assuming

that a inverted list is b bytes long and has b_1 skip pointers. Skip pointers is assumed to occupy four bytes of memory, and each skip pointer skips d_b bytes. Then the expected number of bytes processed is given in Equation 3.14.

$$4b_1 + \min\left(\frac{kd_b}{2}, b\right) \tag{3.14}$$

The reasoning behind the $\min\left(\frac{kd_b}{2}, b\right)$ part of the equation is that one will never decode more than the entire skip list. Also, if the number of documents to be located in the list is larger than the length of the list and therefor the effect of skipping is assumed to be very low, one could choose to ignore skip pointers. The paper summarizes the relationship between b, k, b_1 and the time estimate (unit: number of decoded bytes) shown in Equation 3.15.

$$T(b, k, d_b) = \begin{cases} b & \text{if } k > b \\ 4b_1 + b & \text{if } k \leq b \text{ and } \frac{kd_b}{2} > b \\ 4b_1 + \frac{kd_b}{2} & \text{otherwise} \end{cases} \tag{3.15}$$

The paper points out that b and k have an important relationship. If b is large, then the list is probably a long list (a common word). Many query evaluation strategies evaluate the longest inverted lists late, when the number of interesting documents are low (k is low). When b is low, k is expected to have a higher value. The authors therefore suggest to select the skip lengths (d_b) based on the length of the lists.

In the paper, queries from a TREC query trace were executed and recordings of the length of each list b and the number of matching documents k was used together with Equation 3.15 to select the optimal d_b. In general very short lists does not benefit much from skipping. Lists between 10^4 and 10^5 bytes long works best with very short skips (20-50 bytes). Lists longer than 10^6 bytes benefit from short skips but are best with longer skips. From this the paper chose to consider skips of lengths between 50 and 200 bytes. In experiments it is shown that such a dynamic assignment of skip lengths achieve better query throughput than any static scheme[SC07].

Chapter 4

Overall solution proposals

In this chapter several methods which can be used to efficiently support the requirements of an XML index described in Section 2. Some of these possible solutions will be implemented and tested, implementation details and test results will be shown in Section 5 and 7. Some of the ideas described in the following sections will be left as further work.

The solutions assume an overall index representation like the one described in Section 1.2 where XML-scopes are encoded as path type identifiers and dewey codes for specifying the within-document XML-node. The solution also assumes that the index is a column store with separate columns for: *documentId*, *frequency*, *position* and *scope*.

4.1 Skipping

As mentioned in Section 3.5 skipping is a way to disregard portions of the index while executing queries. During query processing information embedded in the index structure (or in some auxiliary structure) allows for skipping forward in the inverted lists.

4.1.1 Skipping index columns

In general when skipping forward in any of the columns, one would need to skip forward in the other columns. If for instance the scope column skipped n entries the position column will have to skip the same number of entries. However, document identifiers and frequencies columns might skip fewer entries. In fact, the word frequencies within each document will decide how many entries one need to be

skipped in the other columns. In the following, the concept *occurrence identifier* is used. The occurrence identifier represents the ordinal position of a single occurrence within the inverted list. The occurrence identifiers of the document identifier and frequency column is shown in the recursive definition in Equation 4.1.

$$occId_i = \begin{cases} 0 & \text{if } i = 0, \\ occId_{i-1} + f_{i-1} & \text{if } i > 0 \end{cases} \tag{4.1}$$

The occurrence identifier is similar to the *storage key* concept used in for instance *C-store*[SAB+05].

4.1.2 Document identifier skipping

Skipping on document identifier have been discussed in Section 3.5. Skip pointers are embedded into the index structure and used to speed up search. Document identifier skipping allows for quickly calculating the intersection of inverted lists in a document level index.

Assuming that the intersection of two lists for the term "the" and "termite" is to be calculated. The word "the" is a stop-word and should perhaps been filtered out during indexing. But the point is that when intersecting a short list with a very long one, skipping can provide great savings.

Skipping on document identifiers is quite well known and tested and implementations usually encode skip pointers which include the document identifier positioned at the end of the pointer. This way if searching for document identifier x one can traverse the skip pointers as until the value at the end of the pointer is larger than x. More elaborate schemes with several levels of skip pointers are simply recursive applications of the single level approach.

One alternative for storing the skip pointers is to store the pointers together in its own portion of the inverted list. This way all skip pointers are located together. Assuming a query processing method without any early termination like the ones used in [AM04, AM05] it should be safe to assume that one would have to decode all skip pointers. Given enough memory to hold all the pointers in some uncompressed form; one could decode the skip-pointer part first and then use them during query processing. This method has the advantage that one can do binary or interpolative searches with good locality in the skip pointers, thus giving the advantages of multi level skip lists without having to store several level of pointers. The method might be more complicated while building the index, however, the improved locality of the searches might offset that extra cost. Additionally, some compression schemes works better when "similar" data is compressed together. Storing the skip pointers together will probably improve compression effectiveness when those methods are used.

Additionally if the query processing system used is written using the iterator design pattern [GHJV95]; query processing without skipping would cause a lot of invocations of the `next()`[1] method. These are often interface method invocations which might incur some extra overhead for each call. Skipping will potentially replace a series of invocations of the `next()` method of iterators with a single `SkipTo()` invocation. In programming languages where method invocations are expensive the savings related to fewer invocations might be significant. Some smart compilers might in-line calls to the `next()` method, however, *interfaces* calls will often be hard to in-line.

As mentioned in Section 4.1.1 when skipping in one column the other columns need to be aligned on the same occurrence identifier. When skipping on document identifiers frequency, position and scope columns must be aligned on the resulting occurrence identifier. Even if the position column is gap-encoded this should be simple since the column will be positioned at the first occurrence of the resulting document, which in turn is the occurrence where the gap-encoding algorithm for the positions is reset.

4.1.3 Path type skipping

When answering path type filtering queries, one might want to skip to the next occurrence in a scope matching the query. One could do this by looping through each occurrence and return those who matches the query. However, it would probably be more efficient to skip through the scope column until a interesting path type is encountered, and when such a path type is encountered.

Aligning on the occurrence id resulting from such a path type skip is somewhat more difficult than a simple document identifier skip. First one needs to locate the document identifier of the occurrence. The occurrence identifier of this document identifier might be less than or equal to the target occurrence identifier. The document identifier column might also be gap encoded which need to be decoded while searching for the correct document. When this is done, one would also need to skip in the position column. If the position column is gap coded it would first need to be aligned at the first occurrence of the target document, and then moved to the target occurrence id while taking gap coding into account.

Since the scope column is not sorted on the path types, skipping need examine the path type of every entry in the column, or it could use *summaries* encoded into the index. For instance for every 128 scope entry, one could encode the different path types present. These summaries can be either complete summaries of the different path types in the block, or some probabilistic summary such as a Bloom-filter[Blo70].

Bloom filters are bit-vectors of some size associated with a set of hash function. It

[1] Here `next()`, refers to the method which moves the iterator to the next item.

supports two operations, one can add values to the filter and one may query the filter. When adding values, the values is hashed using the hash functions in the filter, and the bits at the hash values positions are set to `true`. When querying the the filter, the same hash functions are used and the hashed positions in the filter is tested. If all the tested position are true, there is a probable match, if one of the positions are false, there are definitely no match. Bloom-filters can produce false positives, but never false negatives when queried.

4.1.3.1 Path type skipping with summaries

One way to support path type skipping is to split the scope column into blocks of some size. These blocks can be given a header, which in turn can store which path types that exists within the block. Skipping then consist of reading the header of one block, skipping the block if the path types present does not match the query.

If the block has some overlap with the query, a fine grained traversal should be performed. It is also possible to include several levels of path type skipping. One could for instance encode the summary for blocks of 256 entries, and then divide the 256 entry block into four sub - blocks, each with their own summaries.

It could be that different summary encodings are better at different levels. One would for instance suspect that the number of distinct path types are higher for high-level (large) blocks. Here, encodings such as a bit-vector can be beneficial. While on small blocks, just listing the different path types could be enough. Also, since the cost of examining a large block is much higher than that of a small block, probabilistic summaries such as bloom-filters might be better fits on the low level blocks.

Since it is not that many different path types even small Bloom filters with for instance 2 or 4 bytes could be efficient. Given a similar summary of the matching path types the overlap can be calculated using a simple bitwise AND operation. If the hash function used to populate the bloom filter is the just the lower 4 or 5 bits of each path type the summary can be built very fast as well.

The way path type identifiers are assigned to each unique path types could cause similar path types to receive consecutive identifiers. Using the lower n bits of the path type identifiers directly as keys in the bloom filter might be to simplistic. If for instance one query matches a set of path types that has consecutive path type identifiers; a small bloom filter could loose some of its effect. Larger bloom filters or more elaborate hash functions could therefor be worthwhile.

Assuming that the set of matching path types for a given query were represented as a bit-vector and the bloom-summary are four bytes long. The summary of a query can be calculated as the running bitwise OR of all the 32 bit integers in the query bit-vector. At least it's imaginable that such operations can be implemented quite effectively.

4.1.4 Combined document identifier and path type skipping

When evaluating structural containment queries as introduced in Section 2.3 one need to perform both a document identifier merge and path type filtering. First one need to get the postings which matches the query. These collections need to be intersected on document identifier in order to be matches. Then one need to merge the within-document-occurrences located within the same scope.

Whether one should skip on document identifier first and then on path type or the reverse could depend on several factors. Consider the case where one is skipping to a very small subset of the available path types, then there will probably be very few matching occurrences. If the index support efficient skipping past non-matching path types. It could be wise to skip on path type first, and then try to skip to the document identifier. When looking for more common path types, document identifiers will probably be the most effective method.

It would be fair to assume that for most cases, document identifier skipping would be very fast, since it searches through fewer values (df versus $df * f_{avg}$), and that the values are sorted. However, if one searches for a very frequent term but for a very infrequent scope, the selectivity of skipping on path type is much higher.

The choice of skipping order might be a task for a query optimizer, and could include extra statistics, both on the properties of the data such as path type frequencies, but also on the index performance characteristics.

In any case, there is no need to align the remaining columns (actually just the position column) on occurrence identifier before both the document identifier and the path types are located. This way some decoding operations could be saved. The ability to wait with such alignment will probably be the only advantage of merging the combined skipping operation into one operation, since the required work for doing the skip operations in the path type column and the document identifier column will be about the same as in the regular case.

4.2 Inverted index compression

As discussed in Section 3.3 there exists several index compression methods with different performance in both decoding speed and compressed index size. One chooses to compress the index in order to save disk and main memory space, but also to make better use of the bandwidth between disk and main memory, and between main memory and the CPU.

Compression and skipping must at some level be aware of each other when used together. Since most compression schemes prohibits random access, one need to include synchronization points, which the skip algorithm skip to. Also as mentioned in Section 4.1.3 columns which is gap coded need some extra consideration when

skipping into them.

In the following sections ideas related to index compression are described. These ideas will be related to the index structures assumed in the previous sections as well as the considerations regarding skipping in the compressed columns of the index.

4.2.1 Modified VByte algorithm

In [Nat09] a modification to the VByte compression method was proposed. By moving the signal bits of the encoded value to the first byte in the codeword only one byte need to be examined in order to decode the value. The different schemes for a four byte value are shown in Figure 4.1. This method is inspired by the dewey control tokens encoding scheme proposed in [HHMW07].

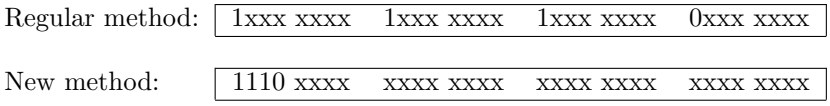

Figure 4.1: A four byte value encoded using two VByte methods.

The idea behind this alternative method is to reduce branching, in the above example only one branch need to be taken based on the first byte value. In the original VByte algorithm one would need to decode the value inside a loop, causing one branch for every byte. Since there are no more than 256 possible single byte values it is possible to pre calculate the interpretation of the first byte and store the results in a single array.

Structure	Description
$Extra[0...256]$	Holds a mapping from byte value to the number of extra bytes.
$Offsets[0...256]$	Holds a mapping from byte value to a special `offset` value.
$Masks[0...8]$	Holds a mapping from codeword length to a bit-mask.

Table 4.1: Branch free VByte datastructures.

It is even possible to decode the new method without using any branches. A few pre calculated data structures are required for this. Those data structures are shown in Table 4.1. If a codeword is encoded in L bytes, then the $Offsets$ mapping for the first byte of that codeword holds a value which is shown in Equation 4.2.

$$Offsets[byte] = MaxVal_{L-1} + remainder \tag{4.2}$$

Here, remainder is the value of the bits not used to encode the length of the codeword. Then when decoding a value starting at position p in the byte array B the steps in Equation 4.3. It can easily be seen that none of the steps contains any branches. One of the primary insights in the super fast *PFor* method described in Section 3.3.1.6 is that the inner loops of the decoding mechanism is without any branches.

However, while branch-free, this new VByte method still retain dependencies between its iterations. However, such data dependencies, are said to be less costly than control dependencies (branches) [ZHNB06].

$$\begin{aligned} E &= Extra[B[p]] \\ value &= Offsets[B[p]] + ((Masks[E] \ \&\ DeRefPointer(B[p+1]) \ << (8-E-1) \\ p &= p + E + 1 \end{aligned} \tag{4.3}$$

There are some issues related to the second step of the decoding algorithm. When dereferencing an eight byte long pointer from position $B[p+1]$ the pointer could span the boundaries of the allocated memory. To avoid this problem the memory buffers used could be padded with eight bytes.

Also, in Equation 4.3 the offsets table is assumed to hold the value of the lower $8-E-1$ bits of the codeword. If one instead encodes the highest $8-L$ bits in the offset table the left shift operation in the second line could be omitted. This will, however, it will make the encoding algorithm a bit more awkward.

From Equation 4.3 the usage of the $Masks$ array is to select those parts of the next eight bytes that are part of the value being decoded. Those masks can be calculated using Equation 4.4. It is clear that a codeword with 0 extra bytes, the mask is 0. While a codeword with eight extra bytes the mask includes all the 64 bits.

$$Masks[i] = (1 << (i * 8)) - 1 \tag{4.4}$$

While the general VByte algorithm can be used to encode and decode arbitrarily large values, this new method is only practical if the range of values to be coded can be determined up front. However, encoding 64-bit integers should be enough for almost any practical use case. At least when used on inverted indexes. On architectures where 32-bit operations are more effective one could use a even more restricted version when the encoded values are known to fit in four extra bytes. Or perhaps for frequencies a version restricted to two extra bytes.

4.2.2 Fine grained skipping

As mentioned in the preceding Sections 4.1 and 4.2 skipping interacts with compression in several areas. Often when skipping into the inverted list there are no skip pointers at the exact location of the value one is searching for. In those cases one will have to traverse the list. This is especially true in the column store index layout assumed in this chapter. When aligning on occurrence identifier skip pointers might not be available and one would need to traverse the list instead.

If it is known in advance that one need to skip k entries in the index, and that one does not need to know any of those values, decoding them would be a waste. This is the case when skipping in a portion of the index that is not gap encoded such as the path type column. Or when it is known that the state of the gap encoding will be reset at the occurrence skipped to which is the case when skipping to the first occurrence of one document in the position column.

Since there is no need to decode the values, it is possible to speed up the traversal. For instance when skipping past values encoded using VByte one can only examine the signal bits of the bytes in the stream. Or even better, if the values are encoded using the new VByte method proposed in Section 4.2.1 one only need to execute Equation 4.5 for each value that is to be skipped.

$$p = p + Extra[B[p]] + 1 \tag{4.5}$$

Similar shortcut traversal opportunities exists in many method. In for instance the word aligned codes described in Section 3.3.3 one can read the number of values encoded in each word by looking at the selector bits. And for the PFor methods described in Section 3.3.1.6 one can easily skip the bit unpacking operations if one knows that none of the values are to be used.

Table 4.2 shows the granularities and high level descriptions of fine-grained skipping capabilities of some of the compression schemes described in Section 3.3.

Most compression schemes employ some sort of variable length encoding where parts of the code specify the length of the code word. Therefore the fine grained skipping described in this section should give some performance gains for almost all methods since the work otherwise put into building the resulting values could be avoided. The only part needed would be the part of the decoding operation which determines the length of the code words. The changes necessary would also probably be small for almost any method since the implementation of a `SkipValue` function would be the same as the `DecodeValue` function where some code are deleted.

The only method among those described in Section 3.3 which does not share this composite representation of code length and code word is the interpolative coding in

Method	Section	Skip granularity	Description
VByte	3.3.2,4.2.1	1	Look at signal bits, or first byte if the method in Section 4.2.1 is used.
Word aligned	3.3.3	1-32	Analyze selector bits, for Relative-10, Carryover-12 and slide maintain state while traversing.
Pfor	3.3.1.6	128	Analyze header for bit-width and number of exceptions, skip through exceptions.
Selectors	3.3.1.5	s_1-s_3 * multiplier	Analyze selector bits, maintain state while traversing.

Table 4.2: Different compression methods and fine-grained skip granularities.

Section 3.3.1.4. This is because the codeword length of a single value is dependent on the decoded values before and after. Huffman codes will also be hard to skip through efficiently.

4.2.3 Column wise dewey encoding

In [Nat09] a dewey code compression scheme where deweys were stored column wise instead of after each other. This way dewey elements at the same positions are encoded together. The idea is that value distribution of the the dewey elements at position i has more in common with other deweys elements at position i in than it has with position $i-1$ and $i+1$ in the same dewey.

Experiments done in [Nat09] calculated the (naïve[2]) entropy for the different dewey positions in three XML collections. The results are shown in Figure 4.2. The results clearly show that all the collections have some positions where the entropy of the value distribution is very low and therefore should be lend the, selves nicely to chunked compression methods.

[2]Naïve in the sense that no other factors than the frequencies of the different dewey values at the various positions were taken into account. A more complex probability distribution for the values taking for instance preceding values and path types into account could have yielded other results.

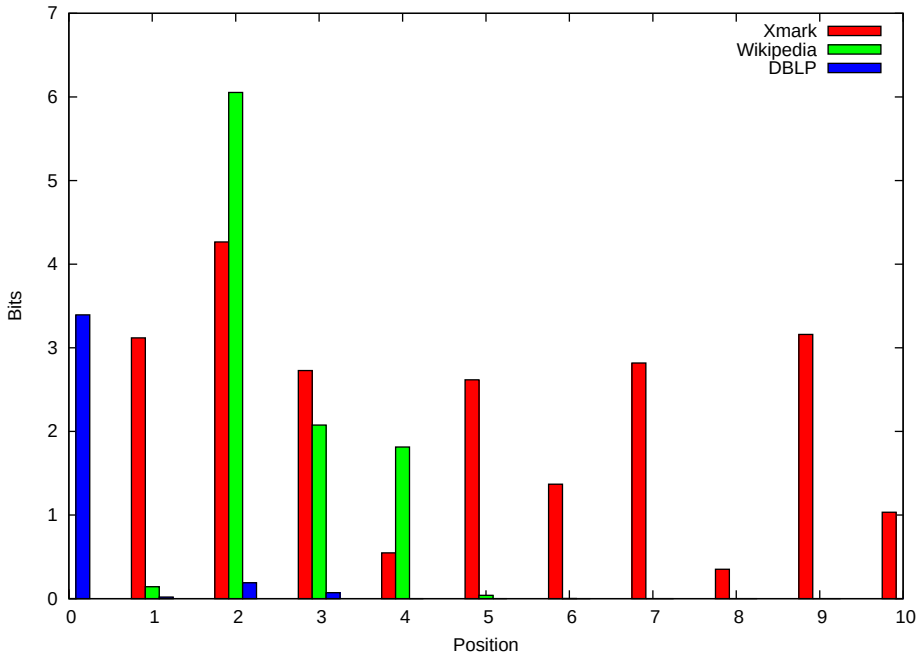

Figure 4.2: Dewey codes entropy in three different XML collections. Since every well formed XML document may only have one root node position zero has been omitted. From [Nat09]

Since most parameterized compression methods works better when the data compressed exhibit somewhat similar value distributes this columnized dewey encoding scheme can improve compressed file size. In Figure 4.3 the idea is illustrated. The figure clearly illustrates that the column wise encoding contains more similar data than the original row wise encoding. Actually the row wise encoding shown in Figure 4.3b resembles the worst case scenario of pointed out by Ahn and Moffat in [AM05]. A sequence $\langle 1, l, 1, l, 1...\rangle$ where l is a rather large value will not be compressed very effectively by a method encoding several values into one unit of compression.

Another opportunity which presents itself with this column wise storage scheme is that in cases where only a limited prefix of the dewey elements is needed for query processing, one does not have to read or decode the remaining columns. This is a more general feature of columnized storage schemes where projections[3] can be evaluated very effectively since one does not have to read all the attributes [SAB+05]. Exploring these opportunities will be left as further work.

[3]The π operator in relational algebra.

$$\langle 0.0.3.2.1.4\rangle, \langle 0.0.3.4.0\rangle, \langle 0.0.3.4.0.1\rangle, \langle 0.0.3.5\rangle$$

(a) Original deweys codes.

$$0, 0, 3, 2, 1, 4, 0, 0, 3, 4, 0, 0, 0, 3, 4, 0, 1, 0, 0, 3, 5$$

(b) Rowwise dewey column data.

$$\langle 0, 0, 0, 0\rangle, \langle 0, 0, 0, 0\rangle, \langle 3, 3, 3, 3\rangle, \langle 2, 4, 4, 5\rangle, \langle 1, 0, 0\rangle, \langle 4, 1\rangle$$

(c) Columnwise dewey column data.

Figure 4.3: Examples of columnized dewey encoding.

4.2.4 Dynamic column wise dewey storage

Column wise dewey storage might be a good choice when there are many elements to be encoded. However, some methods such as *PFor* are unpractical when the number of elements is to small. While row wise storage schemes encode a stream of dewey elements, the column wise method encode the columns, if every dewey is for instance six elements long, the stream of the elements is six times longer than the content of each of the dewey columns. If *Pfor* need to encode for instance 100 elements in order to be effective, there is quite a large window where the row wise encoding has more than 100 elements, and the column wise methods does not. A method which automatically chooses between row wise and column wise dewey storage could boost performance.

4.2.5 Prefix coding for columnwise dewey encoding

In [HHMW07] prefix coding for dewey codes were one of the most effective mechanisms for dewey compression. Used together with other effective methods great compression results will be possible. Prefix coding were experimented with in [Nat09] with mixed results, compression seemed quite good, however, decoding the index was slow. However, it was pointed out that the method tested in [Nat09] were quite suboptimal in terms of utilizing the locality of the value distribution within each dewey column. The method copped off the suppressed prefix of the deweys and encoded the first element in the remaining dewey at position zero. As described in Section 4.2.3 and Figure 4.2, the different columns of the dewey have different value distributions.

One possible way of improving prefix coding is to maintain the position of every dewey element, if a prefix of length three is removed, then one should start encoding the remaining dewey into the fourth column of the column wise dewey storage.

Also in [Nat09] the prefix lengths for the prefix coded versions were encoded using a fixed width binary code. To make the implementation more flexible, one could encode the prefixes as one of the dewey columns, possibly using a *PFor* method.

4.2.6 Path type sorting

Another method proposed in [Nat09] is the path type sorting method. The main insight behind this method is that when sorting the dewey codes on their path type identifier, dewey codes from the same path types will be placed after each other. Such a grouping has several impacts.

First, the value distribution of the deweys from the same path types might be similar. This might especially be true if the index consist of data from several different collections. As shown in Figure 4.2 the entropy of the deweys differs between collections.

Second, a query often matches only a subset of the path types in the index. In this case the dewey codes for each of the matching path types are stored together. This could lead to fewer decoded values if the implementation allows for independent decoding of the dewey groups.

In order to be practical during indexing the path type sorting need to be performed on a chunked basis. A chunk consists of a the occurrences for some number of documents for instance 128 documents as used in [ZLS08, YDS09b, YDS09a, DHYS08]. Then when writing the index, the deweys are sorted using a stable sorting algorithm such as `Counting-Sort`. A stable sorting algorithm will preserve the internal order among elements with the same sort key[CLRS01]. `Counting-Sort` is a $O(n + k)$ stable sorting algorithm, where n is the number of elements to be sorted, and k is the width of the range of the values [Knu98]. The extra overhead from method during compression will be one additional pass over the scope fields in the chunk, and a single pass over the value range. Since the number of different path types is assumed to be quite small the value of k is limited.

The decoding algorithm for a path type sorted inverted list is shown in Algorithm 3. The algorithm uses an auxiliary table (P) of pointers into the sorted dewey array. The table of pointer is constructed by Algorithm 4. Together these two methods are similar to a Counting-Sort algorithm, and could be considered the inverse of the operation used while encoding the file.

Implementations will probably be a bit more elaborate than Algorithm 3 in order to factor in for instance selective decoding. Path type sorting can also be combined with a column wise encoding scheme, the P array should contain pointers into the different dewey columns.

Several variants of this method were mentioned in [Nat09]. One could for instance select the compression method of the dewey codes in one group based on the path types. This way if some path types have very special value distributions one can choose to encode these deweys using a special method. The order of the different path types were also a subject that were discussed. One can for instance sort on

Algorithm 3: Decoding for path type sorting, `Advance` is a function which moves a dewey pointer to the next dewey.

```
Input: Occs occurences.
Data: P mapping from PathTypeId to pointer into dewey storage.
Data: D dewey storage.
1 foreach Occ ∈ Occs do
2 |   DeweyPtr ← P[Occ.PathTypeId]
3 |   Occ.Dewey ← D[DeweyPtr]
4 |   Advance(DeweyPtr)
5 end
```

Algorithm 4: Path type pointer construction, `Keys` is a function which returns the keys in a mapping, `Predecessor` returns the key imidiately before the key passed to it.

```
Input: Occs occurences.
Data: P mapping from PathTypeId to pointer into dewey storage.
Data: C mapping from PathTypeId to the number of occurences with that
      Id.
1 foreach Occ ∈ Occs do
2 |   Increment(C[Occ.PathTypeId])
3 end
4 P[1] ← 0
5 foreach i ∈ Keys(C) do
6 |   P[i] ← C[Predecessor(i)] + P[Predecessor(i)]
7 end
```

the depth of the path types, or the lexical ordering of the different path types. If one choose to sort on the lexical ordering will place paths with the same ancestors together. This could be effective since many queries might match all children of some path. Frequency ordering were also proposed, if the most frequent paths are stored first most queries will read these paths, but probably not the less frequent path types. Example path type orderings are shown in Table 4.3.

One positive side effect from sorting the path types by their frequency is that the C mapping store din the index can be gap coded. This could make the compressed index smaller. Also, sorting by frequency is computationally very easy.

Depth	Lexical	Frequency
`para/sentence/url`	`para`	`para/sentence/`
`para/title/subtitle`	`para/sentence/`	`para`
`para/sentence/`	`para/sentence/url/`	`para/sentence/url/`
`para`	`para/title/subtitle`	`para/title/subtitle`

Table 4.3: Different path type orderings

4.2.6.1 Index building cost

The algorithm used during indexing is described in Algorithm 5. In the for loop in line $1-3$ the path type counts are calculated. Then in line $8-12$ the path type sorting and the mapping from path types to positions in the output are calculated. The P array will at position i contain the offset of the deweys belonging to the ith path type. Also the ordered path types and their counts are written as headers to the index. In line $14-17$ the deweys themselves are written to the index. Using the P array to decide the output positions.

The complexity of Algorithm 5 is dominated by the two passes over the *Occs* collection. However, in the worst case all occurrences contain unique path types the sorting operation might have a cost of $O(n \lg n)$. Therefore the running time of the algorithm is $O(n \lg n)$[4]. In general however, there will be many more occurrences than unique path types and in those cases the two passes over the source collection will dominate the cost.

[4]Sorting in general is $O(n \lg n)$, even though there exists methods which achive better results than $O(n \lg n)$ they usually assume something abouth the values which is to be sorted. If we assue that the number of unique path types is less than $O\left(\frac{n}{\lg n}\right)$ the cost of the sorting step will be $O\left(\frac{n}{\lg n} \lg\left(\frac{n}{\lg n}\right)\right)$ wich in turn is $O(n)$, thus the path type sorting algorithm is $O(n)$ if we assume that the number of uniqe path types is $O\left(\frac{n}{\lg n}\right)$

Algorithm 5: Path type soriting index building procedure.

Input: $Occs$ the occurences to be encoded.
Data: C mapping from PathType to the number of occurences with that path type.
Data: K mapping from PathType to the ordinal position in the sorted path types.
Data: P mapping from ordinal position to output position.

```
1  foreach Occ ∈ Occs do
2  |  IncrementOrAdd(C, Occ.PathType)
3  end
4  P ← new int[UniqueKeys(c)]
5  K ← new int[UniqueKeys(c)]
6  P[0] ← 0
7  ordinal ← 0
8  foreach key ∈ SortedKeys(C) do
9  |  WriteHeaderValue(key, GetCount(C, key))
10 |  P[ordinal] ← P[ordinal + 1] + (GetCount(C, key) ∗ key.DeweyDepth)
11 |  K[key] ← ordinal
12 |  ordinal ← ordinal + 1
13 end
14 foreach Occ ∈ Occs do
15 |  WriteDeweyAt(P[K[Occ.PathType]], Occ)
16 |  P[K[Occ.PathType]] ←
   |  P[K[Occ.PathType]] + Occ.PathType.DeweyDepth
17 end
```

Chapter 5

Implementation

In order to perform experiments, a minimal XML-search engine has been created. This search engine is called *NETing* and will be described in the following sections. The search engine is implemented using the Microsoft .NET framework. NETing is written in the $C\#$ programming language with some components written in $C++$. This search engine is based on the work done in [Nat09].

5.1 Index structures

There are three main index structures used in *NETing*, the dictionary, the scope directory and the inverted index. The dictionary is implemented as a simple hash map from term to a pair: $\langle pointer, documentFrequency \rangle$. The dictionary is not compressed, and is held entirely in main memory during execution.

The scope directory is a data structure which manages a dictionary coded collection of the path types encountered in the indexed document collection. The data structure is similar to that one described in Section 1.2 and [Gri07]. This data structure is also held in main memory and it is not compressed when stored on disk.

The inverted index is a word level inverted index with extra scope information added. Each occurrence of a word is indexed as a triple $\langle pos, pathType, dewey \rangle$. Additionally, each document occurrence is indexed as a single document identifier and a in-document-term-frequency. This way one logical entry in the inverted index will be of the form: $\langle docId, tf, \langle pos, pathType, dewey \rangle^+ \rangle$.

The detailed physical layout of the inverted index is configurable, however, the implementation enforces a column store layout which resembles the one used in

[ZLS08, YDS09b, YDS09a, DHYS08]. This layout allows for high flexibility in terms of available compression methods and allows the various columns to be stored using different storage schemes. The layout of the *NETing* posting file is shown in Figure 5.1.

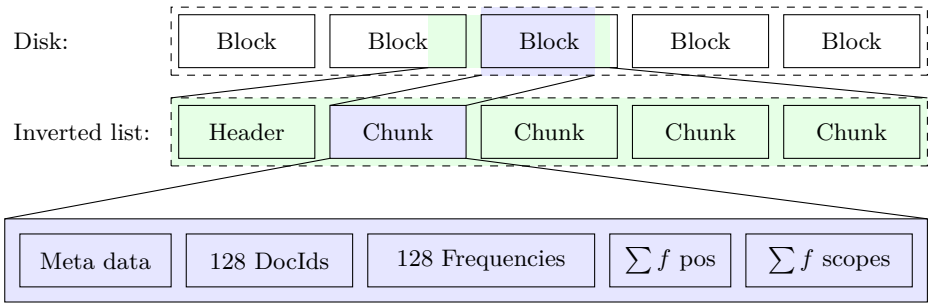

Figure 5.1: Layout of NETing posting file, $\sum f$ refers to the number of occurrences in the chunk, it is equal to the sum of the in-document frequencies.

Each inverted list is divided into several chunks. Each chunk contains the data for 128 documents. Inside one chunk the values of one column (for instance the position column) is stored contiguously. Also each chunk contain some meta data, this meta data contain pointers into the beginning of each of the columns, as well as data which can be used during skipping.

There have been created a common interface for the code which should write and read the posting file. This interface makes it easy to create different column implementations. Each column is allowed to write both meta data and column content. One possible use for the column meta data is for the scope column to write path type summaries used while skipping (Section 4.1).

The posting file is word aligned, each column will start at a word boundary. However, there are no limitation to the alignment of the encoded data within each column. This enforced word alignment might waste some space for very short posting, however the cost of this is assumed to be small.

5.1.1 Buffer manager

As shown in Figure 5.1 the posting file is divided into blocks. Even though the posting file is disk resident these blocks can be cached in main memory. This caching is managed by a component called the *buffer manager*. The buffer manager manages a pool of blocks which can hold portions of the posting file. When answering queries the posting file readers request blocks from the buffer manager instead of issuing the IO operations themselves. The buffers are fixed size, and a

size of $16KB$ has been selected. Each buffer is identified by its position in the file, for the block-id of the block starting at position 0 is 0, while the block-id of the block starting at the offset $64KB$ is 4.

During query processing the operation of the buffer manager is quite simple. When a certain block is requested, the buffer manager checks whether the block already exists in memory, if not it reads the block into memory and returns it to the caller. When the pool of blocks is empty, the buffer manager needs to evict one block which is currently not in use. This is managed through a `Pin()` and `UnPin()` mechanism which maintains a *pin-count* for each block. If a block has a pin count of zero, it is safe to evict the block from memory.

During index building the buffer manager has the additional task of cleaning blocks. A block is marked as dirty if the content of it has been changed. And during indexing that is what happens. New content is written to the buffers, and the buffer manager must write these changes to disk. This writing of the new data is handled by a separate thread. This way the client, in this case the indexer does not need to wait for IO to finish before continuing with the construction of the index.

The buffer manager maintains two queues. The *clean* and *dirty* queue. These queues contain all the buffer with a pin count of zero. When a buffer is unpinned and its pin-count reaches zero it is entered into one of these queues. The dirty queue if the buffer has changes, the clean queue otherwise. The eviction policy is a simple LRU policy where used buffers are entered into the end of the clean queue, and evicted from the front of the clean queue. The tread which cleans blocks take blocks from the dirty queue, writes their changes to disk and enters the buffer into the clean queue.

The mechanisms for manipulating the various queues and the other data structures inside the buffer manager require that some synchronization between threads are done. So in order to allow for better concurrency during query processing the buffer manager is partitioned across several "shards". Each access to the buffer manager is first hashed on its block-id to one such "shard" and processed at that "shard". This reduces the amount of waiting during accesses to the buffer manager.

During the implementation work some experiments were performed. Several different number of shards were tested in order to find the best shard count parameter. The tests consisted of a query trace consisting of 1000 `AND`-queries run over a collection of 300000 Wikipedia articles, good performance were defined as high throughput. On a dual core computer, 16 gave the best results, for a quad core the best performance was achieved with 64 shards. However, the difference between 32 and 64 shards for the quad core computer were very small.

5.1.2 Posting file concepts

The implementation of the posting file introduces a few concepts. These concepts are used in the description of the query processing algorithms introduced later in Section 5.2. The notion of *occurrence identifier* is used on the chunk-level. That is, within each such 128 document chunk, occurrence identifiers uniquely identifies the different occurrences. This means that the range of occurrence identifiers within one chunk is given in Equation 5.1.

$$0 \leq occId < \sum tf_i \tag{5.1}$$

Since each chunk resets the occurrence identifiers, one can skip into the chunk without maintaining state related to the occurrence identifiers.

There are three levels of loading for each column in a chunk. These three levels are summarized in Table 5.1. This three level system exists in order to make sure that the less data is read and decoded when skipping through the posting list. For instance, a position column will start out in the *start* state. And if for instance a skip operation is issued which does not have any hits in the chunk it never has to enter any of the "higher" states. This means that for the position column, no data were decoded and only a small header were loaded.

Level	**Decription**
Start	Header loaded.
Examine	Header decoded for examination.
Ready	Content loaded and ready to be loaded.

Table 5.1: Loading levels for chunks in the inverted index.

The header should as mentioned earlier contain information which makes it possible to discard the chunk column during skip processing. Columns which does not participate in such a skip operation does not need to leave the *start* state. While columns, such as the document id column or the scope column which is needed to process the skip will enter the *examine* state, or the *ready* state if necessary.

Section 4.1.4 introduced the combined path type and document identifier skip operation. Typically if one would execute those two operations as one document identifier skip, and another path type skip one would need to sync the different columns after the first skip. By combining the two skips, one may postpone the sync until the end of the path type skip. For the different load levels. If there are no hits for the second operation then for instance a position column does not need to leave the *start* state.

If the query processing algorithm only uses for instance the document identifier column to identify a match, the other columns will not be *sync*ed before a result

Name	Parameters	Description
`SkipToDocument`	*skipTo*	Moves to the first document identifier $\geq$ *skipTo*.
`SkipToDocumentIdAndPathTypes`	*skipTo*, *pathTypes*	Moves to the first document id $\geq$ *skipTo* which also has a path type in the supplied *pathTypes* collection.
`SkipToScope`	*skipTo*, *depth*, *pathTypes*	Moves to the first occurrence which has a dewey that $\geq$ *skipTo* when looking at a prefix of length *depth* or is at a *documentId* $>$ *skipTo* which also has a path type that in the supplied *pathTypes* collection.

Table 5.2: Skip operations used by the query processing algorithms.

has been found. This way, when processing regular intersection queries the scope column and position column of chunks without any matches will remain in the *start* state.

5.2 Query processing algorithms

The algorithms used to process the queries are built to work with an arbitrary index implementation. The algorithms are based on skip operations supplied by the index implementation and a generic *merge* operation. Three different query processing algorithms have been implemented.

All the algorithms are based on selecting the "highest" list, the one where the next occurrence has the highest document identifier. And then try to skip the other lists to the corresponding occurrence. This is done successively until all lists is positioned on matching occurrences. The algorithms uses a predefined set of skipping-primitives which the index implementation supplies. The primitives are shown in Table 5.2 and are described more in detail in Section 5.2.3.

Keeping the processing algorithms "constant" while varying the column implementations make it easier to compare the different implementations performance.

5.2.1 Intersection queries

Intersection queries are based on a document identifier skip. The inverted lists of all the terms in the query is kept in a binary max-heap. This data-structure provides access to the maximal element in a collection in $O(1)$ time, with $O(\lg n)$ updates [CLRS01]. The maximal element is extracted (the highest document identifier) and then the remaining lists are skipped to this document identifier. If one list is exhausted the algorithm terminates. If the skip operation hits a document identifier "after" the previous maximal value, the algorithm starts all over again.

When all lists ends up on the same document identifier after the skip operations, there is a match. Then all the occurrences within the document in all the lists is returned as the match. This means that the *unit of retrieval* for the intersection queries is all the occurrences for the terms in the query within the matching document. A more formal definition of the structure of the unit of retrieval is shown in Equation 5.2. Here, $\underline{d}$ is the document identifier of the matching document, $\underline{d}$ is underlined as it's a unique key for the result set of the query. That is, there will be at most one result for each document. These results will contain the key $\underline{d}$ and possible several lists of occurrences.

$$\left\langle \underline{d}, (occ^{+})^{+} \right\rangle \tag{5.2}$$

As mentioned earlier in Section 5.1.2 the intersection algorithm will only use the document identifier column during the search operation. The synchronization of the other columns (on occurrence identifier) is therefore postponed until the algorithm knows for sure that there is a match. If the retrieval unit of the algorithm were only the resulting document identifiers, the alignment operation would not have been necessary at all. However, such a restricted version have not been created. The implementation for this thesis assumes that the user might want to employ some ranking algorithm on the results which will use both the word positions and the scope information to deliver ranked results.

The algorithm for answering regular intersection queries is shown in Algorithm 6. It is worth to notice that during the development of the code for the query processing components a *profiler* were used to determine where the execution time of the program were used. This was done in order to select the "hot spots" to focus the tweaking efforts on. The profile revealed that almost all time is spend on skipping and result construction.

5.2.2 Structual containment search

The structural containment algorithm is more complex than the intersection algorithm. Since the results of this algorithm is located within the same document,

Algorithm 6: Intersection query algorithm.

```
   Input: lists inverted lists for all words in the query.
   Data: heap max-heap over the lists based on document-id.
 1 while True do
 2 |   skipTo ← Max(heap)
 3 |   foreach list ∈ Rest(heap) do
 4 |   |   if not SkipToDocument(list, skipTo) then
 5 |   |   |   TerminateQuery
 6 |   |   end
 7 |   |   IncreaseKey(heap, list)
 8 |   |   if list > skipTo then
 9 |   |   |   GotoLine 2
10 |   |   end
11 |   end
12 |   AlignColumns()
13 |   BuildResult()
14 end
```

document level intersection is a subtask of the algorithm. The retrieval unit for the structural containment searches is a scope inside one document, and all the occurrences of the query terms within this scope. The structure of the retrieval unit is more formally defined in Equation 5.3.

$$\langle \underline{d, s}, (occ^{+})^{+} \rangle \tag{5.3}$$

In Equation 5.3 the pair $\underline{d, s}$ is underlined to emphasis the fact that document identifier and scope together is the unique key for the results set. That is there can be multiple results for each document, however, only one for each scope in the document. It also means that if one assume that there is a higher number of distinct scopes at level $L + 1$ than on level L the result of specifying the path portion of the query as for instance `document/paragraph/sentence` instead of `document/paragraph` will result in a possible higher number of result, since every sentence with the required matches would be returned, instead of the matching paragraph. The matching paragraph could contain several matching sentences. However, as a result of this; the returned results from the the query at the `sentence` level would contain fewer word occurrences.

To answer containment queries two preprocessing steps are applied to the set of matching path types. First, since all matches which is "under" the queries path types should be included in the query the union of all the descendants of the query path types should be added to the set of candidate path types. Second, since the prefix of dewey elements need to be analyzed during query processing the prefix of different path types that actually need to match is calculated. This is achieved by

a bread-first-traversal where for each path type pt with children $c \in \text{children}(pt)$ the query depth qd(pathType) for each children is assigned by using Equation 5.4.

$$qd(c) = \min(qd(c), qd(pt)) \tag{5.4}$$

If the search starts with the query path types in ascending depth and in breadth first order the qd array will contain the shortest prefix of each path type which matches the query.

As described in Section 4.1 skipping for containment queries should include skipping on both path types and document identifiers. Variants regarding the order of these operations are mentioned. In the following, these variants are encapsulated in a generic `SkipToDocumentIdAndPathTypes`. The algorithm is a two phased algorithm. First a document match need to be located, this is similar to the general intersection operation, however, with path type restrictions based on the query. It is, however, important to note that a document match does not mean that there is a true match. Occurrences can be located within different scopes, and the path types might not share the required prefix. For instance if one searches in `sentence` scopes, some occurrences may be in a `paragraph/sentence` scope while others are in the `title/sentence` scope. Both scopes will match a `sentence` query, but they are not the same even if their dewey codes match. This top level algorithm is shown in Algorithm 7. It is very similar to the intersection algorithm (Algorithm 6) but differs in the skip method used and the possible multiple calls to `IsTrueMatch` and the possibility for multiple results from each document.

The `IsTrueMatch` step traverses the postings from the different lists trying to locate a collection of occurrences where all the postings are both in matching path types and within the same scope. The path type matching is determined by a longest common prefix algorithm between the path types. The pairwise longest common prefix between path types are cached in a two matrix lcp where $lcp(t1, t2)$ is the longest path type which is a prefix of both of them. To conserve memory space only the upper half of this matrix is calculated, however, the size of the matrix is still $O(n^2)$ where n is the number of distinct path types. There exists algorithms to achieve longest common prefix in constant time with no more than linear extra memory[Gus07][1], however, these would have been very hard to implement, and is out of scope for this thesis. The number of distinct path types are not that high either, so the squared memory cost is a rather low price to pay for fast constant time lcp lookup.

The top generic `IsTrueMatch` algorithm is shown in Algorithm 8 the algorithm uses a `SkipToScope` function in order to align the lists on a matching scope. The list with the scope with the highest lexicographical value is used as the value the other lists are skipped to. This is similar to the methodology used in both document

[1] The problem is the same as solving the lowest common ancestor of the tree containing all the path types.

Algorithm 7: Top level intersection algorithm for structural queries.

```
   Input: lists inverted lists for all words in the query.
   Input: pathTypes lookup structure for all the path types which matches
          the query.
   Data: heap max-heap over the lists based on document-id.
 1 while True do
 2 |   skipTo ← Max(heap)
 3 |   foreach list ∈ Rest(heap) do
 4 |   |   if not SkipToDocumentIdAndPathTypes(list, skipTo, pathTypes)
   |   |   then
 5 |   |   |   TerminateQuery()
 6 |   |   |   return False
 7 |   |   end
 8 |   |   IncreaseKey(heap, list)
 9 |   |   if list > skipTo then
10 |   |   |   GotoLine 2
11 |   |   end
12 |   end
13 |   AlignColumns()
14 |   while IsTrueMatch(pathTypes) do
15 |   |   BuildResult()
16 |   end
17 end
```

level intersection and the top level intersection algorithm for structural queries described in the preceding paragraphs. The skip to scope function has a double return value, the first value is a boolean status flag teller whether the skip operation succeeded. The other value is a boolean flag which tells whether the occurrence that was skipped to is a match or not. It is possible for it to be both. `SkipToScope` skips to the first scope which is greater than or equal to the `skipTo` parameter on the supplied `depth`, if there are no occurrences which is prefix-equal to the `skipTo` parameter, the next occurrence is retrieved, the skip operation succeeds, but there are no match.

Inspection of Algorithm 8 reveals that the `IsTrueMatch` algorithm has three stop criterion. First if one of the inputs are exhausted the algorithm terminates. Second, if one of the inputs skip "beyond" the document of the `skipTo` parameter, the algorithm terminates, it is the top level intersection algorithms responsibility to establish document level matches. The third case is when all the inputs has matches.

Determining the lexicographical highest scope value might be computationally intensive, so, instead the position column is used to determine which list to use as

Algorithm 8: Top level intersection query for structural queries.

```
   Input: lists inverted lists for all words in the query.
   Input: pathTypes lookup structure for all the path types which matches
          the query.
   Data: heap max-heap over the lists based on document-id and dewey order.
 1 while True do
 2 |   skipTo ← Max(heap)
 3 |   depth ← qd[skipTo]
 4 |   foreach list ∈ Rest(heap) do
 5 |   |   (skipOk, matches) ← SkipToScope(list, skipTo, depth)
 6 |   |   if not skipOk then
 7 |   |   |   TerminateQuery()
 8 |   |   |   return False
 9 |   |   end
10 |   |   if list.DocumentId > skipTo.DocumentId then
11 |   |   |   return False
12 |   |   end
13 |   |   IncreaseKey(heap, list)
14 |   |   if not matches then
15 |   |   |   GotoLine 2
16 |   |   end
17 |   end
18 |   AlignColumns()
19 |   return True
20 end
```

the skip-target. Since all the occurrences are drawn from the same document, the positions and the scopes have the same ordering. That is, for to given occurrences i and j with positions p_i and p_j, and scopes s_i and s_j. $s_i \leq s_j$ if and only if $p_i \leq p_j$. Comparing the integer values of the position column is a much simpler operation than comparing more complex dewey codes in the scope column. However, this comes at a price. Since the algorithm uses the values from the position column these values has to be decoded, bringing the position column into the *ready*-state. However, compared with the scope column the position column is very small (and simple), and profiling has revealed that very little time is spent on decoding the position column.

5.2.3 Generic skip support

For easy experimentation and experimental comparability a generic skip interface was created. The three different skip methods were shown in Table 5.2 and used through the algorithms in Section 5.2. In the following sections, the algorithms

used to orchestrate the skip operations on a inverted-list level will be described. The terminology introduced in Section 5.1 where each list is divided into chunks will be key in the following section. At a high level, the generic skip algorithms will focus on quickly loading the correct chunk of the list, and also reduce the amount of data decoded. All while using generic interfaces to the column implementations of each chunk. The idea is that the query processing algorithms should use the generic skip support supplied by the inverted list, while the algorithms described in the following sections will manage the skipping on the chunk-level.

The column implementations have two roles in supporting skip functionality, there are coarse skipping which uses data encoded in the header of the inverted list. A document identifier column can for instance store a collection of entry points here. These entry points will be pairs of document-identifier and file offsets, when answering document identifier skip operations these entry points can give the basis for a coarse level skip. The possibility of storing skip pointers together in the header of the list were described in Section 4.1.

Second the column implementations allow for low level skipping. This is the type of skipping which provides answers of the type, *yes, the value you are looking for is here at position X* or *the value you are looking for is not in this chunk*. Low level skipping uses both in-chunk meta data and the data of the column itself. For instance the document identifier column described in the previous paragraph, could store the largest document identifier in each chunk as part of a chunk-header. Skip operation could examine this header in order to exclude the chunk during processing. Sometimes, for instance when the document identifier the user is skipping to is less than the maximal document identifier in the chunk the values has to be decoded and scanned through in process the skip, this is also handled by the column implementations.

5.2.3.1 Document identifier skipping

The first step in the document identifier skip is to extract an *EntryPoint* from the column implementation. This corresponds to the coarse level skip described earlier. The entry point could as described earlier have been extracted from the list header, but for columns that do not have implemented advanced coarse skipping features the entry point could just be the current state of the index reader.

When at the entry point, the skip algorithms continue by reading the necessary meta-data from each chunk and possibly skipping the chunk based on the content of this meta-data. This functionality is however left to the chunk implementation to handle. The generic skip algorithm will just call `SkipTo` on the document identifier column-chunk and act upon the results from that call.

The `SkipTo` method of a document identifier chunk should return the occurrence identifier of the resulting occurrence, or if there are no match inside the chunk

one occurrence identifier which is larger than the number of occurrences in the chunk. For instance, if there are 432 occurrences in the chunk and the document identifier chunk examines its meta data and concludes that the document identifier the skip operation is targeting must be in a later chunk, it can return 433. The skip algorithm will then continue with the next chunk.

The document skip algorithm does not sync the other columns when it returns. This would have caused extra work in the cases where position and scope data are not needed. An example of such cases is when a intersection query only has partial matches across the lists in the query. The operation to sync the remaining columns is placed in a separate function `SynchColumns` which also were used in the algorithms in Section 5.2.

Algorithm 9 shows the generic document identifier skip algorithm. The loading of the document identifier chunk meta data has been made explicit it order to illustrate that the other columns have no interactions during the algorithm.

Algorithm 9: Generic document identifier skip algorithm.

Input: *docId* document identifier to skip to.
Data: *docIdColumnChunk* the current document identifier chunk.
Data: *chunkMaxOccurrenceId* the largest occurrenceId in the chunk.

```
1 entryPoint ← GetEntryPoint (docId)
2 ReadAt (entryPoint)
3 DecodeDocumentIdChunkMetadata ()
4 occurrenceId ← SkipTo (docIdColumnChunk, docId)
5 while occurrenceId > chunkMaxOccurrenceId do
6 |   MoveNextChunk ()
7 |   DecodeDocumentIdChunkMetadata ()
8 |   occurrenceId ← SkipTo (docIdColumnChunk, docId)
9 end
```

5.2.3.2 Path type skipping

The generic path type skipping method is built around the same template as the document identifier skip operation. First the entry-point is located through some list level meta data managed by the column implementation. Then chunks are processed successively.

Here, as for the document identifier skip operation it is responsibility of the column implementation to supply good entry points as well as en efficient in-chunk `SkipTo` function which for instance can use in-chunk meta data to avoid decompressing chunks which does not match the query.

As discussed in Section 4.1.3 path type skipping is harder than document identifier

skipping because the data is not sorted. Encoding summaries for the path types were proposed as a solution. These summaries can be coded in the list-global header and processed in the `GetEntryPoint` function. Chunk-level summaries are handled by the `SkipTo` function of the column chunk implementation.

Algorithm 10: Generic path type skip algorithm.

```
   Input: candidate the set of candidate path types.
   Data: scopeColumnChunk the current scope column chunk.
   Data: chunkMaxOccurrenceId the largest occurrenceId in the chunk.
1  entryPoint ← GetEntryPoint (candidate)
2  ReadAt (entryPoint)
3  DecodeScopeColumnCunkMetadata ()
4  occurrenceId ← SkipTo (scopeColumnChunk, candidate)
5  while occurrenceId > chunkMaxOccurrenceId do
6  |  MoveNextChunk ()
7  |  DecodeScopeColumnCunkMetadata ()
8  |  occurrenceId ← SkipTo (scopeColumnChunk, candidate)
9  end
```

The generic path type skip algorithm is shown in Algorithm 10. Also here the synchronization with the other columns is left out in order to allow optimizations where only the one column is needed. The implementation of the `SyncColumns` function is quite straight forward, a function which moves the columns to one specific occurrence identifier within the current chunk is called for each of the columns which is out of sync. The implementation of that function is up the column implementation itself. The implementations can range from simply iterating through the items in the chunk until the correct occurrence identifier is reached. However, all the column implementations tested in this thesis have more advanced methods implemented. These methods uses the fine-grained skip capabilities of the compression schemes described in Section 4.2.2.

5.3 Compressed column implementations

All methods tested depends on one or more of three basic methods, these are *VByte*, *Pfor* and *Simple9*. Two implementations exists for *VByte*, the regular method from the literature implemented as described in Section 3.3.2 and the proposed improved method described in Section 4.2.1.

PFor is implemented by using the *NewPFD* method proposed in [YDS09b] and described in Section 3.3.1.7. Exception values are encoded using the *Simple9* method described in Section 3.3.3. The bit packing procedure of *NewPFD* is generated by a python script and contains fully unrolled code for packing 32 values into b bits

wide slots. Similar to the method used in [ZLS08], chunks are padded with zeroes if the number of values is between 100 and 127, however, if there are fewer than 100 values to encode, the values are coded using *VByte*.

The *PFor* and *Simple9* methods are implemented in Visual C++ and can be compiled as both native code and managed code running on the .NET CLR. This will be described in detail in Section 5.4. In the following sections the concrete implementations are described. The details that will be focused on is the layout of the compressed chunks, and how the skip and sync-to-occurrence identifier methods work. There are only three different implementations for the three non-scope columns and these are reused for the different scope column variants.

5.3.1 A note regarding VByte

During implementation some micro benchmarks were performed on the *VByte* implementation. It turned out that for values encoded in one or two bytes, the branch free version which uses a sort of "one-size-fits-all" model to avoid branches was slower than one alternate version. The cost of accessing an (not aligned) eight byte integer made the method slower.

The alternate version selected uses a single branch on the result from the $Offset$ array in Equation 4.3 to select how to decode the value. If the value is less than $MaxVal_1$ the code is 1 byte long, if it is less than $MaxVal_2$ the code is two bytes long and so on. This method is the one that will be used in the experiments.

Evaluation of the branch free version on data where the encoded values need more bytes can be done in further work.

5.3.2 Non scope columns

The non scope columns are *documentId*, *frequency* and *position*. For simplicity the code for managing the *documentId* and *frequency* columns are written in a single component. The column is refered to as the *document* column. This means that there are two column implementations for the three non scope columns, for each compression implementation.

5.3.2.1 Uncompressed implementation

Skipping to $DocumentId$ within the uncompressed chunk representation is handled by a simple binary search within the column. The frequency column does not store the frequencies, instead it stores the running sum of prefixes which in fact equals the definition of the occurrence identifier. The frequency can be inferred by taking the difference between one occurrence identifier and the previous one. Storing the

occurrence identifiers instead of the frequencies themselves means that when skipping on document identifier no post processing is required after the binary search, if the frequencies were stored one would need to iterate through the frequencies in order to devise the occurrence id the skip operation stopped at. Also, when syncing the document column to one occurrence id a single binary search in the $frequency$ column is enough.

The uncompressed implementation of the position column is trivial. It is a single array of integers. No skip operations is supported, but syncing to occurrence identifier consists of simply accessing the $occurrenceId$'th position in the array.

Uncompressed column implementation does not encode any header information. The document identifier is encoded as four byte unsigned integers, while both the frequency and position column is encoded using two byte unsigned integers.

5.3.2.2 VByte implementations

Even though there are two *VByte* implementations they are discussed as one in the following paragraphs. The implementation for the *VByte* document column encodes one single piece of header information. The largest occurrence identifier in the chunk is encoded in the header in order to exclude the chunk quickly when processing document identifier skipping. The document identifiers are gap encoded and when syncing to on specific occurrence identifier the elements in the document id and frequency columns are traversed together while calculating the running sum, the running sum of the frequencies adds up to the occurrence identifier.

The position column is also gap encoded so that the first occurrence in each document are encoded with its original value and subsequent occurrences are encoded with gaps. When syncing to one occurrence identifier the implementation first skips to the first occurrence identifier of the document that the target occurrence identifier is located in, and then continues forward while calculating the running sum. The first skip can utilize the fine grained skipping optimization for *VByte* described in Section 4.2.2.

5.3.2.3 PFor implementations

PFor methods are only capable of decompressing chunks of 128 elements at the time, this means that it has to decompress the entire document identifier column of a chunk if the values are to be retrieved. To avoid doing this to often, the largest document identifier in the chunk is encoded in the header. Just as with the *VByte* implementation this header value are examined in order to exclude the block if the document identifier that is skipped to is not present in the chunk. The document column is gap coded and the frequency column store the frequencies of the documents and the running sum need to be calculated in order to maintain the

occurrence identifier during both skipping and column syncing.

The position column is gap encoded the same way as the *VByte* version and syncing to occurrence identifier is handled the same way here as for *VByte*. The fine level skipping mechanism supported by the *PFor* method is used to move the decoder to the first occurrence of the document that is being skipped into and then the running sum is calculated while iterating forward. It should be pointed out that the granularity of the fine grained skipping supported by *PFor* is quite coarse. Only blocks of 128 elements can be skipped over, however, skipping such a block is much faster than in the *VByte* case.

5.3.3 Scope column

The scope column is the most complex column to compress and it has the most different number of implementations. Most of the versions are implemented using *PFor* compression as the compression primitive, but also uncompressed and *VByte* implementations exist. In the following sections each of the implementations will be described.

5.3.3.1 Uncompressed scope column

The uncompressed column is simply a long array of two byte unsigned integers. There are two pointers into this array. The `pathTypeIdPointer` which starts in first position in the array and moves one step for each occurrence that is retrieved. The second pointer is the `deweyPointer` which starts at the first position after the path types and together with the path type (which defines the dewey depth) this pointer specifies the dewey of the occurrences. For each retrieved occurrence `deweyPointer` is incremented by the dewey depth of the previous path type. Both path type skipping and occurrence identifier syncing are managed by scanning through occurrences and maintaining the `deweyPointer` with respect to the dewey depths.

5.3.3.2 VByte scope column

Both *VByte* versions share the same scope column implementation. The only thing that is different is the compression algorithm it self and how fine grained skipping is implemented. The layout of the column is quite simple, first all the path types are stored after each other compressed using *VByte*, then all the dewey elements are stored after each other.

The scope column for *VByte* has one value in the header of the chunk, the number of extra bytes in the path type section of the column. Here extra bytes is defined based on the insight that no value can be coded in less than one byte when using

VByte, extra bytes is the difference between the number of bytes used to compress the path types in the chunk and the number of path types. If all path types are encoded in one byte the value will be zero. This is a small optimization which means that the header field can be compressed better than if the total number of bytes used to encode the path types were to be stored.

Also here two pointers are used to point at the locations where the values are to be decoded from. The `pathTypePointer` starts at the first position in the column chunk, and is moved according to the number of bytes used for each of the encoded values. The `deweyPointer` starts at the byte number number of occurrences plus the number of extra bytes stored in the header. The dewey pointer is moved forward while decoding the dewey elements required for each occurrence. The path type decide how many dewey elements that are retrieved and *VByte* decides how many bytes to move the pointer for each of the values.

Path type skipping and occurrence identifier syncing are done by traversing the path types while maintaining the total number of dewey elements passed. Then the dewey column is skipped the correct number of entries forward using the fine grained skip ability of the *VByte* implementation.

5.3.3.3 PFor scope column

The *PFor* scope column encodes the path types in chunks of 128 entries first, and continues with the deweys after that last path type value has been encoded. For each block of 128 path types the total sum of all the dewey depths are stored in the header. This meta data is used during column syncing. Also a pointer to the beginning of the dewey data is stored in the header.

During decoding only one path type and dewey block is decoded at the time, when the next occurrence is retrieved the current block might be exhausted and a new one will be decoded, this can happen for both path types between occurrences and for deweys between dewey elements.

Path type skipping is handled by traversing the path types encoded in the chunk, the running sum of dewey depths are maintained in order to skip the dewey column forward when the search locates a match. When a match is found the dewey part of the column is skipped forward using the fine grained skipping mechanism supported by *PFor* just like for the *PFor* implemented position column. Syncing to an occurrence identifier uses the dewey depth sum information located in the header to skip past path type blocks as well. The only path type block that require decompression and dewey depth summation is the block containing the target occurrence identifier. The dewey depth summation can either be performed from the beginning of the last block of the sync operation or from the end of the block, in the latter case the addition to the total number of skipped dewey elements will be the difference between block total depth and the reverse sum.

5.3.3.4 Path type sorted PFor scope column

The path type sorted version of *PFor* is a implementation of the ideas from Section 4.2.6. The path types is stored in the original order, but the dewey codes are stored in a order which is sorted by their path type. This way deweys coming from the same path types are placed together. The ordering chosen for this implementation is based on the path type frequencies in the chunk. Dewey codes for the most frequent path types are place first in the dewey portion of the column.

This implementation has quite a lot of different header values. First, a *VByte* encoded pointer to the beginning of the dewey portion is stored compressed. Then the number of unique path types in the chunk is stored compressed with *VByte*. After these two *VByte* encoded values a series of values compressed with *Simple9* is stored. These are the path type identifiers and their respective in-chunk frequencies.

Path type identifiers are not stored in their original form, they are stored as the respective path types rank in the frequency ordered collection of path types. Thus, the range of possible values in the path type portion of the chunk is narrowed, allowing for better compression.

The path types are managed the same way as the regular *PFor* implementation. One chunk is decoded into memory and when that chunk is exhausted a new one is decoded. Instead of a single `deweyPointer` this method maintains a pointer array with one pointer for each path type. The values of these pointers are the ordinal dewey offset into the sorted sequence of dewey values. When decoding the occurrences the pointer for the current path type is extracted. By shifting the pointer seven bits[2] to the left the "block"-number is extracted. The the block number is three, the third block of dewey values need to be decoded. Of course every block only need to be decoded once, when the block first is accessed, it is decoded and stored in a array indexed by the block number. It could be that the blocks are not accessed in their block number order, in those cases the blocks before the requested block are skipped past in order to locate the target block. However, each block is at most skipped past once and skipping past blocks compressed with *PFor* can be done very fast.

The dewey pointer array resembles the P array in Section 4.2.6.

The path type counts stored in the chunk header is used as a path type summary during path type skipping. If the candidate set does not overlap with the path types present in the header the content of the block can be discarded without decompression. If there are occurrences of candidate path types in the chunk the path type are traversed until an occurrence is observed. During this traversal the number of occurrences of each path type is counted in order to advance the dewey pointers.

[2] Or dividing by 128

In the early phase of development a method where the path type frequencies were decreased as their occurrences were returned from the chunk were experimented with. This would have allowed queries that were searching for a very infrequent path type to skip past the rest of a chunk if the occurrences of that path type had been exhausted earlier. However, the code required to make this work during skipping an occurrence identifier syncing made it slower than it was without.

Occurrence identifier syncing is achieved in much of the same way as the path type skipping. The occurrences of the different path types is counted during the traversal and the dewey pointers are advanced according to the encountered path types. It should, however, be noted that this differs from the way the regular *PFor* column handles syncing. There are no skipping in the dewey column involved here. Since the decoding of the dewey column is done on-demand it is enough to move the pointers.

5.3.3.5 Column wise PFor scope column

The column wise *PFor* implementation stores the data in a somewhat different way than the regular implementation. One chunk contains a repeated structure of 128 path type identifiers followed by the dewey codes for those path types compressed column wise. That is one *PFor* block with path type identifiers and *maxDepth PFor* blocks with the dewey values. Here *maxDepth* refers to the depth of the deepest path type among the 128 for each such block. Since deweys are of variable length, the deepest dewey columns will contain less than 128 elements. In order to decode these values correctly the number of elements in the columns which contain less than 100 elements need to be encoded in the index. 100 elements is the limit where the *PFor* implementation will default back to a *VByte* implementation instead of padding the values to fill all the entire 128 slots in a *PFor* block. For each block of 128 path types; the number of "full" blocks is encoded in a single byte[3] and then the number of elements in the remaining blocks is encoded as single byte values (since they all are less than 100 using even a single byte is wasteful, however, to keep the implementation simple (and byte align) full bytes are used for those values.

During decoding only a single block is decoded at once. Pointers into the dewey columns are maintained and for each occurrence the prefix of the pointers corresponding to the deweys depth are moved to the next position in the decoded values.

Path type skipping is handled by traversing the path type portion of each 128 block, if a match is found the deweys for that block are decoded. If there are no match the dewey columns can easily be skipped by first skipping the "full"-blocks number of *PFor* encoded blocks and then skip past the remaining *VByte* encoded values.

[3]This limits the deweys to a maximal depth of 256.

Syncing on occurrence identifier is done in a similar manner, except that the path type blocks may also be skipped past. Only the target block will need to be decoded and then the dewey pointers need to be moved according to the depth of the deweys inside the current block.

5.3.3.6 Dynamic column wise PFor scope column

Column wise PFor seems fine, however, when used on columns with less than 100 occurrences the *VByte* fall-back mechanism for *PFor* will be used for on all the dewey columns. This is quite waste full. To alleviate this a "meta-method" is introduced. The dynamic column wise *PFor* scope column will dynamically select between regular *PFor* and the column wise version based on the number of occurrences in the chunk. If there are more than 100 occurrences, the column wise version will be used, if not the regular version will be used.

This dynamic solution should both improve compression effectiveness and performance.

5.3.3.7 Prefix coded column wise PFor scope column

Prefix coding deweys share some resemblance with gap encoded position columns. The common prefix among dewey codes in one document is suppressed. Instead of storing the entire dewey, the index contains one integer determining the length of the shared prefix and the remaining of the dewey elements. The method applied here stores the remaining deweys column wise in a way that retains the position of the dewey element. This means that there are *maxDepth* dewey columns. If prefix coding removes a prefix of three from one dewey code, the first value that is encoded for that dewey will be placed in the fourth dewey column.

As an optimization the first dewey code in one document is also prefix coded, by assuming that there are an occurrence "the minus-one-occurrence" in every document with a dewey consisting of only zero dewey elements some prefix coding can be applied to every occurrence in the index.

The prefix coded scope column stores the path types in a contiguous portion of the chunk. Then all the prefix lengths follow, these are also encoded using *PFor*. For each block of 128 prefixes the number of dewey elements in each of the columns are encoded. This information is used when skipping past dewey codes. Since prefix coding affects the number of elements in the first columns of the deweys it is easier to encode the number of elements remaining than to iterate through the prefix values and calculate the counts based on path type depths and prefix lengths. The column counts are encoded using single byte values since all the values are less than or equal to 128.

Last the dewey columns are stored, here all dewey elements at position i is stored in one column. During decoding only one group of 128 values will be decoded from each column at the time. Then that block is exhausted, a now one is loaded. The choice of storing the columns together as opposed to the 128-occurrences at the time method used in the plain column wise *PFor* method is because the prefix coding removed many values from the beginning of the deweys, and therefore more values would be encoded using *VByte* as the fall-back mechanism. By encoding more data together these effects will be smaller.

In the header of the prefix coded *PFor* scope column chunk the offsets to the prefixes and each of the dewey columns is encoded using *VByte*. Also the total count for each of the dewey columns are stored. This is done in order to be able to decompress the last block in each of the columns. The count is needed in order to select the *VByte* fall-back scheme if there are less than 100 remaining values.

Decoding is handled by loading one block of 128 path types and prefix-lengths at the time. For each new document that is decoded the dewey object is initialized with only zero element values. Then for each occurrence the path type is retrieved and the prefix length. Then a loop goes through the values from the end of the suppressed prefix and to the end of the dewey and adds values from the dewey columns. The dewey columns also only store one decoded chunk of values for each of the columns.

Path type skipping is handled by traversing the path types until a match is located. Then the algorithm skips past the prefixes until the first occurrence of the document where the correct pat type were found, and from there iterates through the deweys. The column-counts encoded together with the prefixes is needed in order to propagate the dewey columns correctly when the skip operation finishes.

Occurrence identifier synchronization is handled the same way as path type skipping, except that the path type blocks are skipped over instead.

5.3.3.8 Path type sorted column wise PFor scope column

This method is a combination of the column wise dewey storage method and the path type sorted method. The same type of ordering is used (frequency ordered), and the same type of header information is included (unique path types, frequencies). In addition, the header contains the offsets to the start of each of the columns. This method uses the column layout of the prefix coded method, where all dewey elements at position i are stored in the ith dewey column.

The same method for lazily on-demand loading the different blocks in the dewey column is used. Pointers are shifted seven bits to the left to get the block number, then the block is either already in memory, or it is decoded into memory. If the pointer is to a block which is not the *next* block in the dewey-column it is skipped to in order to obtain the offset of the block. Note that each block will be at most

skipped past once.

As opposed to the original path type sorted version there is a two-dimensional array of dewey pointers, one pointer for each (path type, dewey column) combination. However, if one interpret the rows in this matrix of pointers as a single pointer into the column wise dewey storage the matrix will resemble the P array described in Section 4.2.6.

Path type skipping is handled by first looking at the path type summary in the header of the chunk, then if there are a match the path type portion of the chunk column is scanned, looking for a matching path type. A count for each of the encountered path types are kept in order to propagate the dewey pointers correctly after the skip. Since the dewey blocks are loaded on-demand no other action need to be taken than to update the dewey pointers.

Synchronization to a given occurrence identifier is done in a way very similar to the skipping. The path type portion of the chunk column is traversed and the counts for each of the path types are aggregated in order to move the dewey pointers just before the operation terminates.

5.3.4 A note regarding SkipToScope implementations

Recall that the `SkipToScope` function that was part of the structural containment algorithm described in Section 5.2.2 and Algorithm 8. This method is used to locate a structural match by comparing a prefix of the dewey codes. In the problem statement in Section 2.5 dewey prefix comparison were named as one of the key areas for high performance *XML*-queries. However, during implementation and profiling, a naive (brute force) method showed itself as fast enough to make the time spent inside that skip function negligible.

This is probably because the test collection used contain documents that is too small. The set of candidate occurrences remaining after a document identifier and path type match is small enough to be processed naively. However, provided large enough documents many interesting optimizations could have been made.

The prefix coded columns could for instance use the encoded prefix lengths to iterate through occurrences as long as $prefix \geq pos_{mismatch}$ if the dewey code differed at position $pos_{mismatch}$. All column wise methods could for instance traverse the results down the columns, searching for dewey elements that match. Comparing deweys in their compressed form could also be experimented with, this is the "state of the art" method with *VByte*, but it might be possible to achieve similar functionality in *PFor* methods. Further work could try to implement these optimizations and run tests with larger documents.

5.4 Unmanaged code

The Microsoft .NET platform provides several ways to call "unmanaged" code from within "managed" functions. The terminology "managed" and "unmanaged" refers to code running on top of the *common language runtime* and code executing machine code directly on the computers hardware. Unmanaged code runs without the memory management and security features offered by the CLR, but possible higher and more predictable performance.

The common language runtime provides the possibility of running programs written in different programming languages together. Is is for instance possible to write a program in C# and use libraries written in Visual $C++$. The programs are compiled into an immediate form called *MSIL*[4] which is executed by the CLR. The different CLR languages provides similar feature sets, however, the language that is the most expressive, and that provide the most advanced features is Visual $C++$. The minimal search engine used in this project is written using *C#*, however, since Visual $C++$ provides very simple integration of unmanaged code the *PFor* and *Simple9* algorithms were implemented in Visual $C++$ and are used as a library

A description of several ways to call unmanaged code from Visual $C++$ is found in [Mic10]. One of the methods are *mixed assemblies*. These are assemblies (a DLL) containing both managed and unmanaged. When compiling Visual $C++$ code with the correct command line arguments[5] the code generated will be managed code. However, by adding the pragma-directive `#pragma unmanaged` in front of a function, that function will be compiled as native unmanaged code. The compiler will manage all the bureaucracy related to the call out of the managed code and into the unmanaged function.

It should, however, be mentioned that all functions are managed by default, therefore one needs to apply the pragma-directive to all functions that are to be called from other unmanaged functions. If the Visual C++ code is simple switching between managed and unmanaged will be easy. For the experiments aiming at testing the possible performance gains from "going native" the code can just be compiled without the pragmas.

There are performance implications of calling unmanaged functions from within the managed runtime. Therefore care has been taken to reduce the number of calls. The code therefore contains methods like `SkipMany` which skips through several *PFor* encoded blocks. The managed client can call this bulk-method instead of issuing several calls to a single `Skip` function.

The interoperability features of the *CLR* makes the both the *C#* code calling the *PFor* libraries look just like any other static method call, and the Visual C++ code just calls regular functions.

[4] Microsoft Intermediate Language
[5] The `/CLR` argument

5.5 Coding details

High performance requires both good algorithm and good code. In the following sections some of the details encountered during the implementation of *NETing*. Since many of the details explained here were discovered and implemented in a non-finished test system; the exact performance impact is hard to describe. Measuring the exact performance gains would not applied to the finished system that the experiments in Chapter 6 will be executed on. However, some guidance regarding the performance implications will be provided.

This part of the thesis could be considered the "tips and tricks" section. Even though some of the details are well known to developers of high performance code the author has not been able to locate similar information in textbooks. There surely exists tutorials and text books on writing high performance *C*, *C#* and *Java* code. The message in most on-line tutorials of "high performance Java" or "high performance *C#*" usually state something like: *don't create new objects, garbage collection will kill performance*, *avoid exsessive boxing/unboxing*, *method calls are expensive* and *delegates are even more expesive*[6]. All these insights are sound and one need to keep those in mind when developing high performance code. In the following sections some *concrete* examples on how to adhere to some of these standards will be provided.

5.5.1 Object reuse to avoid high garbage collection costs

One of the easiest ways to destroy high performing code is to allocate a lot of objects. Object creation in *C#* is quite fast, however, every object created need to be collected by the garbage collector. If there are many objects to be collected, garbage collection pauses could consume most of the execution time of the program.

In *NETing* there is an object of the class `PostingFileEnumerator` which handles the interaction between query processing components and the posting file. All skip operations are handled through it, as well as plain traversal of the inverted list. When first implemented, a test run just iterating over the index revealed that 70% of the execution time were used in garbage collection[7].

By running a memory profiler, it was shown that a high percentage of the object creation came from `PostingFileEnumerator`. Also, quite some CPU time was used to build these quite complex objects. To counter this, a `PostingFileEnumerator` pool was created. This pool allowed `PostingFileEnumerators` to be reused between queries. This rather simple change made time in garbage collector drop to 20% which still is high, compared to final version which uses 2% of the time.

[6] delegates is the *C#* version of type safe function pointers.

[7] The garbage collection statistics were collected using the built-in tool `perfmon.exe` which is included in most recent releases of the Windows operating system.

Pooling objects can help alleviate garbage collection problems, and it is especially effective when the creation of new such object is a complex and time consuming process, then the object creation cost can be amortized across multiple uses. However, for very simple objects, the cost of accessing the pool and possible synchronizing threads at that pool might come at at too high a price.

For small and simple objects "holder" objects can be used. For instance, the output of the `PostingFileEnumerator` is *occurrence* objects. These occurrence objects contain four fields $\langle doc, freq, pos, scope \rangle$ and there are several thousand of them. Instead of creating new objects for each occurrence, the same object is returned every time. In stead of allocated the holder object is mutated. This cuts down on the number of allocations with thousand of objects for each query, and is one of the methods used to keep the garbage collection time as low as 2% in the final version of *NETing*.

5.5.2 Buffer invalidation by sequence number

In languages like Java and *C#*, memory is cleared when it is allocated. If you allocate an array of integers, all values are 0. This is different from the default behavior of ancient languages such as *C* when allocated memory can contain almost anything. However, when reusing objects as described in Section 5.5.1, arrays will contain the data from the previous use, which can be almost anything.

One can write code that will zero-out the arrays, and this is in many cases the correct thing to do. However, for many of the data structures used internally in *NETing* column implementations such as the lazy loading dewey columns in the *path type sorted column wise scope column*, clearing arrays are not always a good use of CPU resources. Clearing a piece of an array is a $O(n)$ operation, if the array is long, the operation takes longer to complete.

For lazy loading dewey columns a bit vector is used to determine if a dewey block is loaded into memory or not. When the object using the bit vector is cleared. This is a very fast operation, however, if the bit vector is going to be used for a very short list, some of the clearing work would be redundant. The query will never hit the tail of the vector. Also, the operation is $O(n)$ indicating that if the bit vector is long, which it could be if the object previously had been used for a very long scope column, the operation will take some time.

One approach which gets rid of the clearing of the vector is to associate a sequence number to the "session" of the reused object. Each time the scope chunk reader object is reused, the sequence number is increased by one. The bit vector is replaced by an array of integers which stores the last sequence number where the block was loaded into memory or not. During processing $status[blockNum] = True$ operations are replaced with $status[blockNum] = currentSeq$. Clearing is now reduced to a $O(1)$ operation since it only consists of incrementing the sequence

number.

Of course there are some extra memory cost of storing an integer array instead of a bit vector, however, there is no need to use four or eight byte integers for this. One can use a single byte unsigned integer to store the sequence number and then do some extra clearing each time the sequence number counter wraps around. Still one would need to measure the effect of such changes in order to decide whether to use them or not.

In the *NETing* case, changes from clearing to invalidation by sequence number gave a slight increase queries per second for the path type sorted column wise *PFor* index implementation. The throughput went up from 135 to 139 on the query trace used during implementation. Profiling also revealed that less time was spent in the method which prepared the chunk column implementation to decode the index.

5.5.3 Loop optimizations

Loop optimization is a well known field within compilers theory and a quick web search reveals large amounts of resources[8]. In the context of *NETing*, loop optimizations apply to a rather simple algorithmic aspects. In Section 5.3.3, skipping and occurrence identifier syncing is described for the different scope column implementations. The general theme for the column wise scope columns is to traverse the path type column and move the dewey pointers according to the path types encountered.

There are at least two ways of doing this. The simplest is to move the dewey pointers once for each path type traversed, somewhat like described in Algorithm 11. This algorithm moves the dewey pointers for each of the path types encountered.

Algorithm 11: Simple scope column traversal.

```
1 foreach pathType ∈ SelectedPathTypes() do
2 |   MoveDeweyPointers(pathType.Depth)
3 end
```

For column wise dewey representations, Algorithm 11 has a running time of $O(nk)$ where n is the number of path types and k is the average depth of the path types. This can be improved by aggregating the depths of the path types encountered. And then move the dewey pointers once instead of for each path type. Algorithm 12 does this.

Line $6 - 9$ in Algorithm 12 transforms the depth counts array into an array which contains the number of position to move in each column. The insight is that initially

[8] `http://www.google.com/search?q=loop+optimization`

Algorithm 12: Aggregated scope column traversal.

```
   Data: depthCounts array where the path type depth counts are aggregated
 1 total ← 0
 2 foreach pathType ∈ SelectedPathTypes() do
 3 |   Increment(total)
 4 |   Increment(depthCounts[pathType.Depth])
 5 end
 6 depthCounts[0] ← total
 7 for i ∈ 1...maxDepth do
 8 |   depthCounts[i] ← depthCounts[i − 1] − depthCounts[i]
 9 end
10 MoveDeweyPointers(depthCounts)
```

position i in the array contains the number of elements ending after i columns. The algorithm starts with the total number of occurrences at position zero and for each position it subtracts the number of finished occurrences.

The running time of this algorithm is $O(n+k)$, assuming that the `MoveDeweyPointers` is a $O(k)$ operation. By transforming some of the loops within the scope column implementations a 10% gain in throughput on a test done during development was measured.

Applying the same principals to the prefix coded scope column is slightly more complex, one does not only need to keep track of the depth of the deweys, but also the prefixes of the encountered deweys. However, the principal is the same, and it will transform the running time from $O(nk)$ to $O(n + k)$. One could note that the depth of the deweys is probably limited and that k actually is more like $O(1)$, however, both algorithms have little extra work so the constant factors in the $O(n + k)$ are also low, making it preferable even if k is small.

5.6 NETing statistics

The *NETing* search engine has been created in the specialization project in fall 2009 and finished in this master thesis January - June 2010. Most of the code is written in *C#*, with some parts, mainly the compression method *PFor* and *Simple9* implemented in Visual $C++$. The statistics for the *NETing* codebase is shown in Table 5.3. The $C++$ code also includes the bit-packing of code generated by a script. This amounts to approximately 3000 lines of code. The open source program `cloc`[9].

[9] `Cloc` version 1.51 to be exact, `http://cloc.sourceforge.net/`

Lines of code (C#):	16575
Lines of code (C++):	3476

Table 5.3: NETing codebase statistics

If not present on some kind of CDROM bundled with this thesis the source code for *NETing* can be obtained by contacting the author[10].

[10] ola.natvig@gmail.com

Chapter 6

Design of experiment

The following chapter will describe the experiments which are to be carried out. The experiments are designed to explore the performance characteristics of a selected subset of the methods proposed in Chapter 4. Tests will be performed by running regular list intersection queries (AND-queries) and structural containment queries since these were the two query types selected in the concrete problem statement (Section 2.5).

6.1 Test collection and query trace

The collection to be used in the experiments is a collection of XML encoded Wikipedia articles. The collection contains 382 thousand articles. The documents have tags for sentences, paragraphs, titles, persons, companies, locations among others. The total size of the collection is 1.57 gigabytes. Statistics regarding the Wikipedia collection are shown in Table 6.1.

Dewey elements have a very skewed value distribution. The lower element values come in far greater numbers than the larger ones. Even though there exists high dewey element values as shown in Table 6.1, Figure 6.1 draws a clear picture that if one place a limit at 100 almost all dewey element values falls below it. More

Distinct number of path types:	238
Max dewey depth:	9
Max dewey element value:	8259

Table 6.1: Wikipedia collection statistics.

than 80% of all dewey elements are less than three. One of the reason for this very skewed distribution is the dewey numbering scheme itself. If there, within one scope, is to exist a child node with the value 2, there has to be another child with the value 1 first. This means that there can be no more twos than ones. Since there are many scopes, there must be many zeros and ones, and as scopes with high fan-outs are less common, there will be fewer threes and fours and so on.

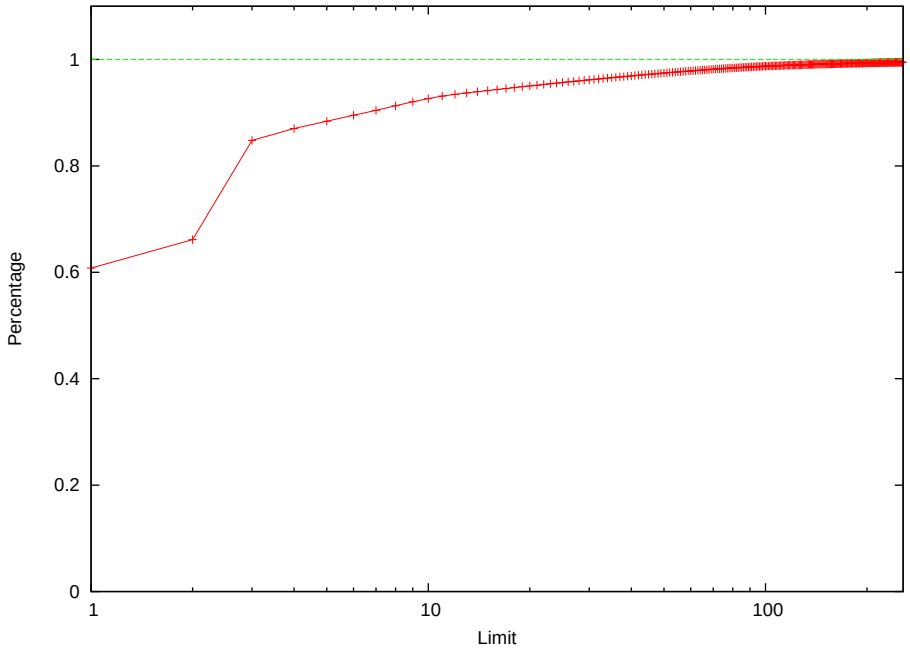

Figure 6.1: Value distribution for the dewey elements in the Wikipedia collection. The plot show the percentage of dewey values that has a value less than or equal to *limit*. The green line show the 100% line.

The basis for the query traces which is to be executes is the 100000 queries in the *TREC 06* efficiency tasks. Since these queries are not designed to match the Wikipedia collection in any way, some preprocessing has been done. In Section 2.5 two types of queries were selected: intersection queries and structural containment queries. In the following sections the process of generating the test query traces from the TREC queries will be described.

6.1.1 Intersection queries

The TREC query trace contains phrases. To generate realistic structural containment queries from the trace each word in each phrase is treated as a term. A query

which intersects all the lists is then constructed and executed once.

If the constructed query has matches in the index it is written to the query trace for intersection queries. Together with the query, the number of results is recorded. This way, it is possible when running experiments to only select those queries where the number of results is within some range.

6.1.2 Structural containment search queries

The same interpretation of the TREC query is used for structural containment searches. Words are extracted and assigned as terms to a query configured as a structural containment search. Since some path types are more common than others, several different path types need to be used.

Four different path types have been selected. `paragraph`, `sentence`, `title` and `company`. With `paragraph` path types, all path types containing a paragraph label is addressed. Almost all word occurrences in the corpus occur within paragraph path types and also within `sentence` path types. It is clear that there are a significant overlap between `paragraph` and `sentence` path types. However, while they both match almost all occurrences, `paragraph` and `sentence` path types are different in that many structural containment hits on the paragraph level will not be matches on the sentence level. `sentence` queries require a deeper matching of the scopes.

Recall that in Section 5.2.2 the retrieval unit for structural containment queries was defined as the triplet $\langle d, s, (occ^+)^+ \rangle$. For a `paragraph` query, the $(occ^+)^+$ part of the retrieved unit will contain all the matching occurrences under the `paragraph` node. For a sentence query, there will be fewer occurrences. But possible additional retrieved triplets.

This can be generalized to that a query which specifies constraints on a deeper level will return results containing fewer occurrences, but possible additional results.

Queries constructed with the `title` path types will hit fewer occurrences than the `person`, `paragraph` and `sentence` queries, but more than the rather exotic path types containing a `company` label. The occurrence statistics of the different path types is shown in Table 6.2.

6.2 Experiment methodology

Since the search engine used in this thesis is running on the Microsoft .NET Common Language Runtime, some extra care need to be taken when executing performance tests. Programs written for the CLR is not compiled into machine executable

Path label	Matching path types	Matching occurrences
`paragraph`	229	150 427 296 (99.92%)
`sentence`	215	150 427 216 (99.92%)
`title`	51	3 147 336 (2.09%)
`person`	39	9 667 327 (6.42%)
`company`	34	1 424 789 (0.95%)

Table 6.2: Occurrence statistics for the four selected path types.

code, instead they are compiled into an intermediate code which is executed on top of a virtual machine. The virtual machines supply memory management through garbage collection, and just-in-time compilation of intermediate code to machine executable code. The performance implications of both garbage collection and just-in-time compilation are hard to predict prior to execution.

Experiments done in [Nat09] showed that if one runs one experiment several times the performance stabilizes very quickly. Therefore one can run the experiments for instance ten times each and then selects the one execution with the best timing results. This gives the just-in-time compiler the ability to optimize the code and generate machine code, as well as disk caches to warm up.

In [Gri07] experiments were performed on software written in *Java*. Java also uses just-in-time compilation and garbage collection. Here the experiments were also run several times, but not a fixed number of iterations. Instead the experiments were executed until the difference between subsequent runs were under some threshold (e.g. 2%). It would be equally correct to do the experiments in this thesis this way, however, a sufficiently large and fixed number of iterations (such as five) were shown to be effective for the CLR in [Nat09].

6.2.1 Main memory resident index

The focus of this thesis is at compression. Compression has impacts in many ways for query processing, one of them is disk transfer. However, as memory prices decline, and 64-bit hardware and operative systems open up for several gigabytes of main memory on search engine servers. It will be practical to hold indexes in main memory. Independent from any such trend. When measuring such black-box characteristics as latency and throughput, the best picture of how the different compression schemes perform against present it self when the compression scheme account for most of the work load. Therefore tests should be executed with the indexes residing in main memory. This is achieved by setting the memory limit of the buffer manager (Section 5.1.1) high enough to cache the entire index in memory. The first execution of each experiment will then populate the buffer manager with the blocks needed throughout the entire experiments. Subsequent experiment will then be performed with the index in main memory.

6.2.2 Multi-threaded tests

Modern computers have the ability to execute several task concurrently. The impact of the different compression schemes when running multi threaded programs should be examined. There is limited bandwidth between main memory and the CPU(s), and it could be that some of the methods which produce larger indexes will be hit harder with this limited bandwidth than the methods with smaller indexes. Experiments should be executed with different levels of concurrency to assess these issues.

The parallelization of the code is limited to running multiple queries at the time. There are no inter-query parallelization implemented.

6.2.3 Different query classes

As stated earlier in Section 6.1 there will be several query traces to execute. However, those query traces are not homogeneous. There might be queries with for instance very different number of terms and results. There might be that a query with very many results has different performance characteristics than queries with only a few results. Experiments should try to gain some insight into the performance implications of both number of terms in the query, and the number of results.

6.3 Compression method labeling scheme

Since several different compression schemes are to be tested, and some of them has very long names a system of labeling the methods need to be made. The labeling scheme is approximately the same as the one used in [Nat09] and consists of short labels combined with dots. One example is **PTS.PFor** with means a path type sorted *PFor* compression scheme. The labels are summarized in Table 6.3.

Tag	**Meaning**
VB	Regular *VByte* encoding.
NVB	The new *VByte* encoding introduced in Section 4.2.1.
PFor	*PFor* encoding.
CPFor	Column wise *PFor* encoding as described in Section 4.2.3.
PF	Prefix coding as described in Section 4.2.5.
PTS	Path type soring as described in Section 4.2.6.
D	Dynamic scope column compression as described in Section 4.2.4.

Table 6.3: Labels for compression schemes.

6.4 Compression schemes to test

Among the many methods described in Section 3.3 and Chapter 4 a few candidates need to be selected for testing.

Among the regular index compression schemes the simplest and probably most popular method *VByte* (Section 3.3.2) is a relevant choice. *VByte* is relevant because the method is widely used, and it has been shown as among fastest of all the single value methods shown in Table 3.5. Additionally this thesis discusses modifications to this method which could be interesting to observe the value of (if any). *VByte* could be considered a baseline method in these experiments.

Another method which someone might called the "state of the art" method is the *PFOR* methods (Section 3.3.1.6) and the *NewPFD*/*OptPFD* variants of this method. Also, previous research has shown it to be extremely fast [ZHNB06, YDS09b, ZLS08]. *PFOR* and its variants are the fastest methods in the "chunk of values" column in Table 3.5[ZLS08].

In addition experiments should be performed on a uncompressed index to see if compression actually is worthwhile.

From the scope specific methods proposed in Chapter 4.2 both path type sorting and columnized dewey storage should be tested. However, since *VByte* is a context-free method, in that it does not care whether the data compressed follows a tight value distribution or not; column wise dewey storage will not be tested with *VByte*.

The dynamic column wise method will also be tested, it should perform similar to *CPFor* on long lists, and *PFor* on short lists, hopefully with a performance that comes close to $\max(CPfor, PFor)$ for different queries.

Even though *VByte* could benefit path type sorting since it would allow selective dewey decompression for structural containment queries it will not be prioritized. Experiments done in [Nat09] showed that the index built using *VByte* already is substantially larger than indexes built with *PFor* methods. Adding path type sorting to a *VByte* column would have increased the size of the index even more. The choice of letting the *VByte* implementation stay simple has therefore been made. Also, the experiments performed in [Nat09] showed that *VByte* methods were significantly slower than PFor methods.

The different index configurations selected for testing is shown in Figure 6.4.

Method	Description
Raw	No compression.
VB	*VByte* compressed index.
NVB	The new *VByte* implementation proposed in Section 4.2.1.
PFor	Simple PFor compressed index.
PTS.PFor	Path type sorted PFor index.
CPfor	Column wise PFor index.
D.CPfor	Dynamic column wise PFor index.
PF.CPfor	Prefix coded column wise index.
PTS.CPfor	Path type sorted column wise PFor index.

Table 6.4: Index configurations selected for testing.

6.5 Performance measurements

The experiments done will execute several different queries, for each such experiments, the following measurements will be done. The *throughput* for the given query log and a given index implementation is defined as the total number of queries executed divided by the total time of the experiment. The unit of the *throughput* measurements is *queries per second*. *Throughput* describes the overall query processing capacity of the system.

Additionally the average *latency* will be measured. The time to answer each query will be measured and an average will be calculated. The *latency* describes how quickly the system can answer a given query. The unit for *latency* is *seconds* or *seconds per query*, but the values will hopefully be in the millisecond ranges.

The ratio between *throughput* and *latency* will probably not be constant as throughput increases (or decreases). Factors such as thread-synchronization in some data structures will probably affect the latency as more threads are used to execute queries.

Latency will mostly be focused on in the experiments where different number of threads are used to answer queries. In those cases the development of the measured latency as throughput (hopefully) increases will show how the threads interfere with each other and affect overall query latency.

6.6 Concrete experiment plan

In the following sections a concrete plan for the experiments will be enumerated. The default configuration of experiments are that the experiments will be run concurrently with four query-threads. The experiments will be run on a computer

with a Quad-core processor so this corresponds to one thread for each processor core. The default structural containment search queries are by default evaluated by a method which skips on document identifier first, and path types second.

6.6.1 Test regarding the impacts of number of results

By running experiments with queries with different number of results, insight into the performance implications of the size of the result set can be measured. When preprocessing the query traces in Section 6.1 the number of results for each query were recorded. Groups of width 10 is used and five groups for each query trace. The experiments to be executed are shown in Table 6.5.

Id	Queries	Description
1	Intersection queries	Grouped on #results, group width 10, five groups.
2	Path, sentence	Grouped on #results, group width 10, five groups.

Table 6.5: Experiments with number of results as variable.

The experiments in Table 6.5 will be the basis for the general discussion of the results for the different proposed schemes.

6.6.2 Native code implications

To test if the implementing the *PFor* compression method in native code boosts performance experiment 1 and 2 should be executed with the implementation compiled both as native and managed code. The performance for each of the *PFor* based methods should be compared.

6.6.3 Test impact of concurrency

The test computer has a quad core CPU. When testing with different numbers of concurrent threads will among others show how memory bandwidth affects performance. Therefore, tests with $1, 2, 4$ and 8 concurrent threads should be executed. Since the query processing algorithms should be orthogonal to concurrency levels these tests could be executed on only one type of queries. Since there are many more intersection queries those are selected for this tests. The experiments to be executed in order to assess concurrency issues are shown in Table 6.6.

Id	Queries	Description
3	Intersection queries	Single query execution thread.
4	Intersection queries	Two query execution threads.
5	Intersection queries	Four query execution threads.
6	Intersection queries	Eight query execution threads.

Table 6.6: Experiments with different levels of concurrency.

6.6.4 Structural containment queries, skip order

In Section 4.1 combined path type and document identifier skipping were discussed. The order of which one chooses to skip were introduced as a possible query-optimization technique. Section 4.1 suggests that for path types with few matches, it could be effective to skip with path types first. Since a procedure based on document id skipping could cause a high number of false-positive matches.

The experiments listed in Table 6.7 are selected in order to measure how the two different skip strategies perform on structural containment queries where the occurrence statistics differ.

Id	Queries	Description
7	Path, sentence	Skip to document identifier first.
8	Path, sentence	Skip to path types first.
9	Path, title	Skip to document identifier first.
10	Path, title	Skip to path types first.
11	Path, person	Skip to document identifier first.
12	Path, person	Skip to path types first.
13	Path, company	Skip to document identifier first.
14	Path, company	Skip to path types first.

Table 6.7: Experiments with different skip order.

Chapter 7

Results

In this chapter the results from the experiments described in Chapter 6 will be presented and commented on. Also a short summary of the results from the pre-project ([Nat09]) will be provided to describe some of the previous results with many of the same methods that have been developed and refined in this thesis. While the main focus for the pre project was indexing and compressed index size. This thesis has focused on query processing, therefore there will be little information on the indexing speed of the different algorithms. However, since most methods were tested in the pre project, hints on indexing speed could be found there.

Method	**Scope column**	**Other**	**Total**
Raw	2100750	688247	2788998
VB/NVB	1061903	369353	1431257
PFor	616546	305089	921635
PTS.PFor	613768	305109	918878
CPFor	480474	305012	785486
PF.CPfor	438536	304972	743508
PTS.CPFor	450639	305017	755656
D.CPfor.100	454484	305032	759517
D.CPfor.64	410520	305032	715552
D.CPfor.32	383385	305028	688413
D.CPfor.16	**376160**	305025	**681185**
D.CPfor.8	376802	305024	681826

Table 7.1: Compressed index sizes (in KB)

Since the size of the compressed index could be a interesting part of the analysis

of the query performance the sizes will be provided in Table 7.1.

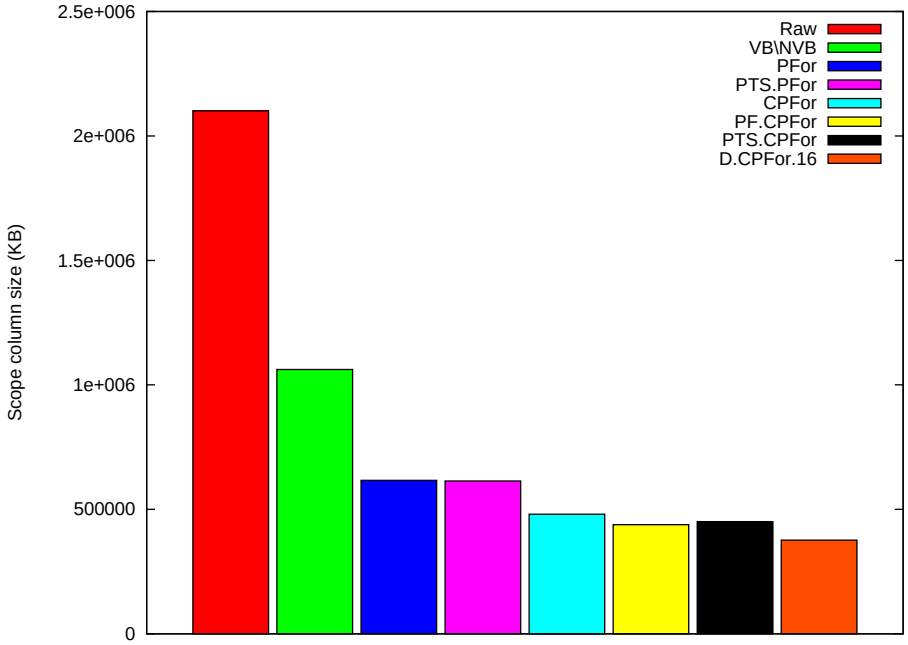

Figure 7.1: Scope column size, Wikipedia collection index. Thesis version.

Since all the *PFor* method share the same implementation for the non-scope columns one would assume that the values in the *other* column in Table 7.1 should contain equal values. However, included in those numbers are meta data stored in the index, for instance are column offsets within one chunk encoded using VByte. For the methods which produce smaller files, these pointers are "shorter" and can therefore, in some cases be coded in fewer bytes. *NETing* does not allow a portion of the chunk headers to span buffer boundaries, therefore some bytes will be lost when there are not room to start a new posting list at the end of a buffer. This may also affect the *other* column in Table 7.1 between methods which do have the same implementations of the columns, but different size of the scope column.

The size of the scope columns used during the following experiments are shown in Figure 7.1.

7.1 PFor cuttoff parameter tuning

The literature suggests a cutoff parameter for *PFor* at 100 elements [ZLS08]. However, this might not be the optimal case for all data. For instance, by looking

at Figure 4.2 one can see that the two first columns of the dewey codes for the Wikipedia collection have very low entropy. When the dewey values in a specific column are placed together they may be compressed very efficiently. And while *VByte* needs to use at least one byte for each value, *PFor* can compress the values is much less than one byte. In the extreme case where all values are equal, PFor can compress 128 values using only four bytes for the *header* of the compressed block. If this is the case, it would be space economical to compress a sequence of only five values.

It is clear that the fixed cutoff of 100 proposed in the literature may be wasteful for some sequences. Therefore experiments with the dynamic column wise *PFor* method were performed with different cutoff values. These experiments were not specified in the experimental plan, however, since the difference in compressed index size from a cutoff of 100 to 16 is as much as 17% measuring the query performance will be interesting.

The size of the *other* portion of the index increased as the cutoff parameter was decreased. A choice to keep the cutoff parameter for the three non-scope columns fixed at 100 was made. Experiments with variations here could also have been made, however, the focus of this thesis is on the scope column.

It might be that the smallest index is not the fastest index, after all an index with a low cutoff value will perform a *PFor* decode operation meant for 128 values to decode as little as 16 values. It might be that this big-block-decode operation takes to much time to be effective. However, *PFor* can be fitted to process blocks as small as 32 items which may be more suitable. The experiments described in the subsequent sections have all been performed with a 128-values *PFor* implementation.

Further work could investigate how dynamically selecting *PFor* cutoff values for different types of data affect compressed index size and query performance. It clearly has the potential at the compressed index level, and with variable block size *PFor* query processing speed could also increase.

The results from all the six different *D.CPfor* methods will not be presented in full here, however, the full result data will be available in Appendix A.

7.2 Previous results from pre project

The focus of the pre-project in [Nat09] was on indexing speed and compressed index size, however, measurements were also done on the decompression speed of the different index implementations. The implementation done in the pre project has been used as a starting point for the experiments done in this project, however, most components have been seriously re-factored and the current code base is much more optimized for high speed query processing. While in the initial implementation the ability to create stacked-configurations of the index was more of a focus. In the

initial *NETing* implementation one could add prefix coding to any scope column by simply changing the configuration. This configurability is nice, but it will in many cases sacrifice performance.

The configurability of the first *NETing* version meant that it was easy to perform many experiments, therefore, [Nat09] ran experiments on more than twelve different combinations of column implementations. The variants included *prefix coding* (Section 4.2.5), *path type sorting* (Section 4.2.6) and *column wise dewey encoding* (Section 4.2.3) which all also will be tested in this thesis.

In addition a simple method called *first-child-tagging* where a single bit was used to encode if the last element in the dewey was a "first child" (.0 or .1 depending on indexing). Also experiments were done on a dictionary method where parts of dewey codes were coded from a dictionary.

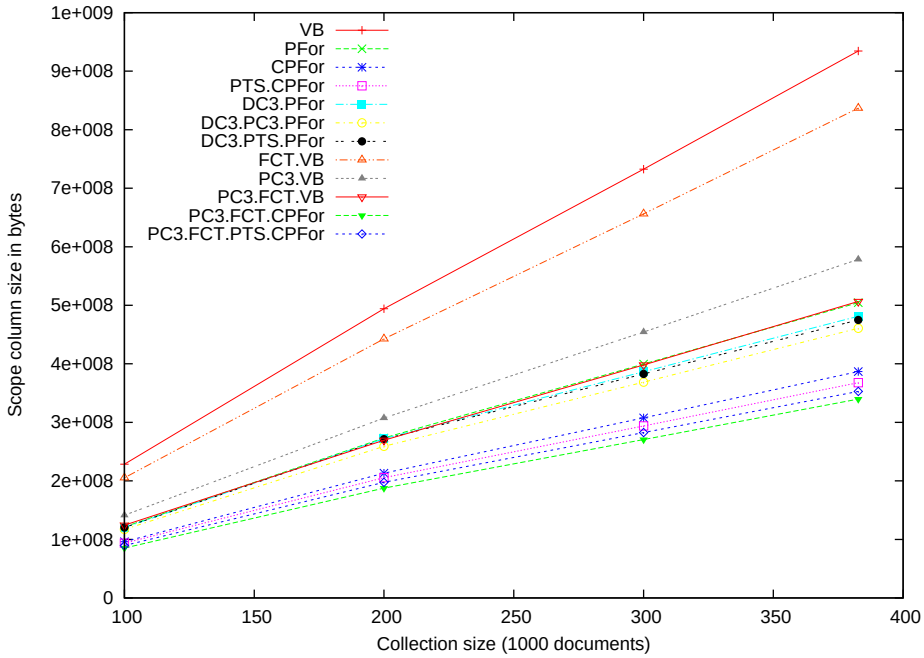

Figure 7.2: Scope column size, Wikipedia collection. From [Nat09].

When looking at the compressed index size methods with prefix coding, column wise dewey encoding and path type sorting performed the best. A plot of a selection of the methods tested is shown in in Figure 7.2. The labels for the different methods are quite similar to the labeling scheme used in this thesis (Section 6.3), but in addition the DCx tag means a dictionary encoding which chunks of three dewey elements. FCT mans first child tagging, and the PCx tag means prefix coding with x bits allocated to store the prefix length.

The results shown in Figure 7.2 resemble the ones presented in Table 7.1. Prefix coding gives improvements for both the current index and the one in the pre project, as does the path type sorting method. *VByte* is also shown as significantly inferior to the *PFor* methods, however, in [Nat09] the prefix coded and first child tagged VByte version came close to the simples *PFor* method.

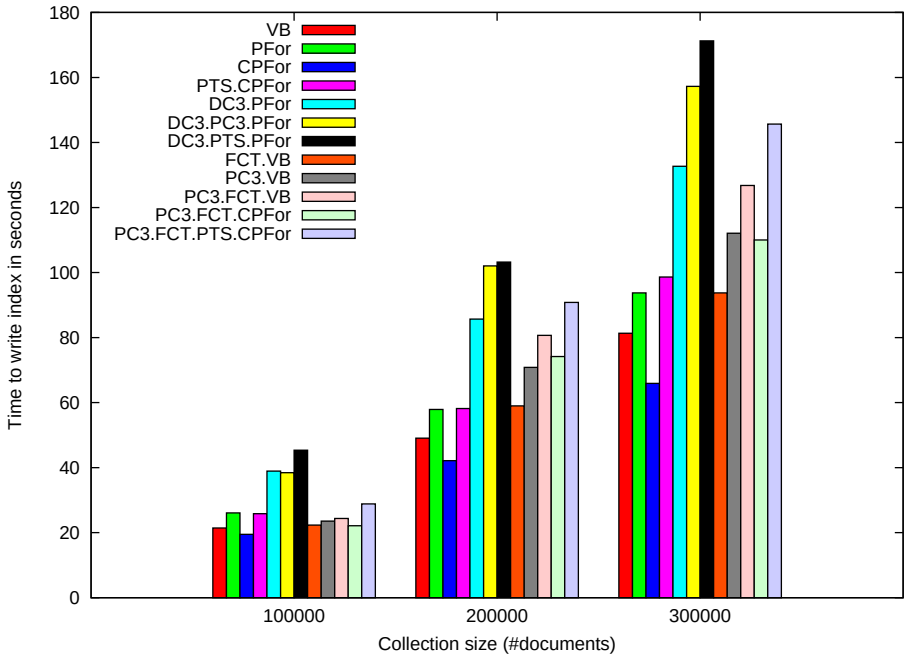

Figure 7.3: Index write time for 100, 200 and 300 000 wikipedia documents. From [Nat09]

In Figure 7.3 the time to build the index using several of the method tested in [Nat09] is shown. It shows that the simpler methods (less prefix coding and so on) allow for the fastest indexing. This could have been due to the perhaps too flexible implementation in the initial version of *NETing*, but partly also because of the added complexity of for instance prefix coding. The fastest method was the simple column wise *PFor*, probably both because of the speed of *PFor* in general, and the fact that there are less data to be written than for many of the competitors. The simple *VByte* method was also quite fast. Path type sorting came at a cost but, perhaps not a high one as expected, considering that there are both extra meta data to be written and extra word done during the sorting phase.

The decompression experiments in [Nat09] were quite simplistic. The time to read all the inverted lists for each entry in the dictionary were recorded. The time was recorded from within the index system so that it was possible to decide how

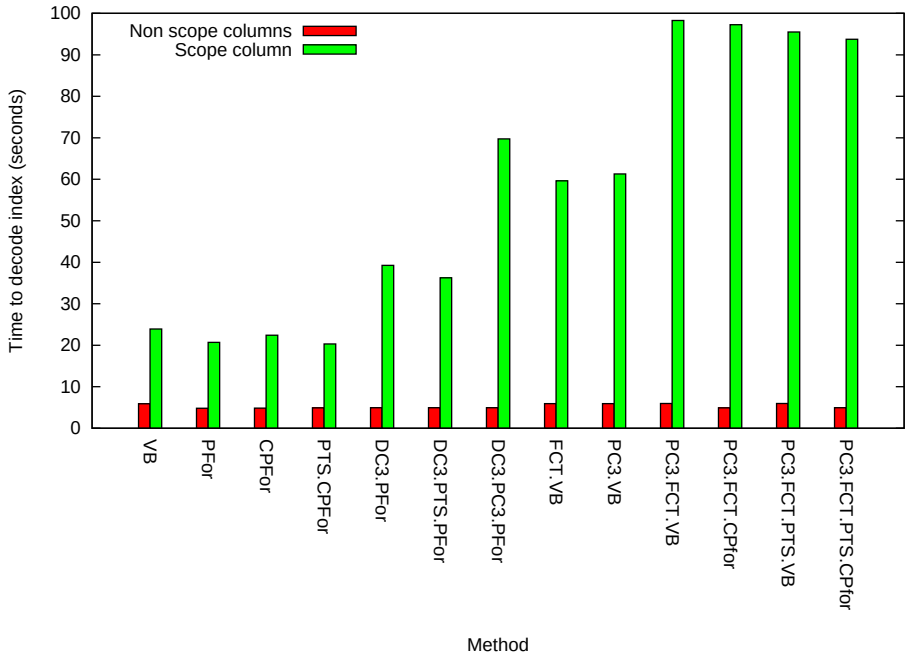

Figure 7.4: Index decoding time for 382 737 wikipedia documents. From [Nat09].

much time that was spent on the scope column and the other columns. While simplistic, this method gave insight into the shear decompression capacity of the methods. However, some of the more composite methods might have suffered from the unoptimized implementation. However, the results shown in Figure 7.4 show[1] that for the non-scope columns *PFor* is 19% faster than *VByte* and that column wise *PFor* both path type sorted and regular are among the fastest methods. However, so was *VByte*.

[1] even though the scale of the graphs makes it hard to see.

7.3 Experiment results

In the following sections the results from the experiments will be displayed and commented on. The experiments were run on a Dell Optiplex 755 with a Intel® Core™ 2 Quad CPU (Q6700 @ 2.66GHz). The test computer had 8 GB of memory, and a single 500GB Hitatchi 7200 RPM hard drive with 16 MB Cache. The operating system was Microsoft Windows Server® 2008 R2. The installed .NET runtime was .NET 4.0.

7.3.1 Impact of number of results

The first two experiments were preformed with queries grouped on their number of results. This experiments were designed to show differences between the implementations when the result sets size increased. The throughput for intersection queries at different result set size intervals are shown in Figure 7.5. The number of queries which were executed for each of the intervals is shown in Table 7.2

	1-10	**11-20**	**21-30**	**31-40**	**41-50**
Queries:	29 666	6 901	3 549	2 255	1 580

Table 7.2: Number of queries used in intersection query experiment.

The results in Figure 7.5 can be grouped in roughly four groups, the uncompressed index with 500 QPS for the shortest results, and 400 for the longer ones. The two *VByte* variants with a little less than 800 QPS for the shortest results and around 600 QPS for the longest. *PFor* methods with a little more than 900 QPS for short results and 750 QPS for the longest, and the column wise *PFor* methods whit a QPS range for 1050 to approximately 800.

Within the four groups there are variations, most notably the column wise path type sorted which consistently has lower throughput than the other column wise methods. The reason for this is probably the more complex implementation, all in all the path type sorted versions include some additional book keeping during decoding. Also, since this is intersection queries, none of the features of the path type sorted index can be utilized to speed up querying. There are for instance no need for decompressing only the dewey codes for some specific path types, and there are no path type skipping which may use the path type summary encoded in chunk header.

The fastest method is one of the dynamic column wise method with a cutoff of 8, 16 or 32 depending on the result set size. This means that one could probably select the one with 16 as a cutoff value, even though this diverges from the literature standard

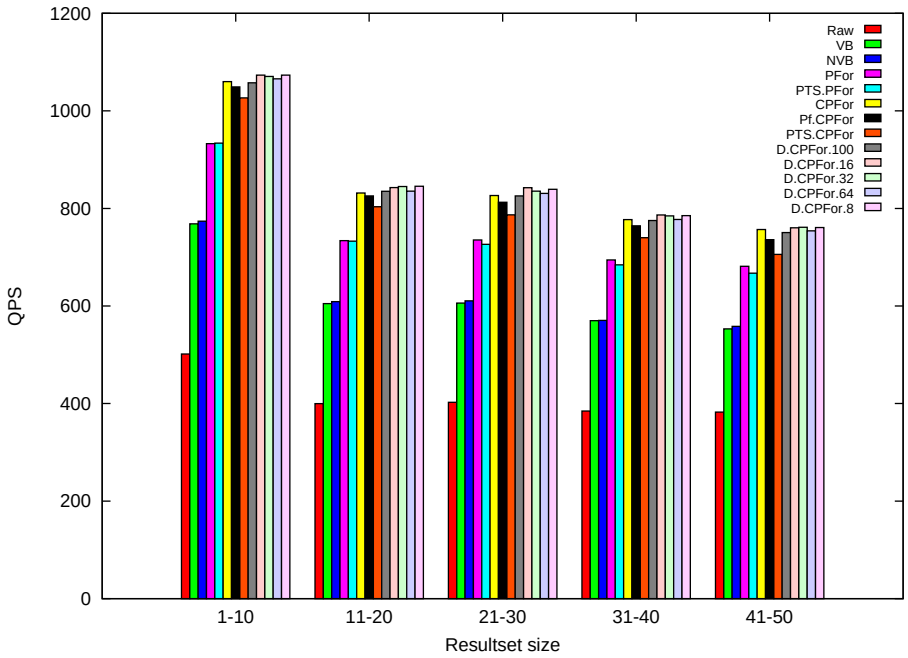

Figure 7.5: Throughput for intersection queries at different result set sizes.

of 100. Again, implementations could try to select the cutoff value dynamically to achieve both better query performance and smaller files.

The new version of *VByte* is faster than the one in the literature. However, the differences are not spectacular. It could be that the gains of the new version is due to its more effective skipping and not only due to the unrolled decoding implementation. It should be noted, that for *VByte* encoded index the scope column represents 75% of the total index size.

Knowing that there are quite few distinct path types, and that the most common ones have low path type identifiers, one can assume that almost all path type entries are the index is encoded in one byte. Furthermore, as shown in Table 6.1 the vast majority of the individual dewey elements can be coded in one byte using *VByte*[2]. This leads to the conclusion that almost all of the values which account for 75% of the index size is encoded in one byte.

If one analyses the implementation of both the regular *VByte* and the final version that ended up as the one being tested in *NETing* (Section 5.3.1) both handle single byte encoded values with one branch. That is, the new version is not as superior in the single byte version as it would be for multi byte values. However, the fact that

[2]They have a value 127 or less.

the performance is slightly better in these experiments is good news with respect to the performance when encoding columns with higher values.

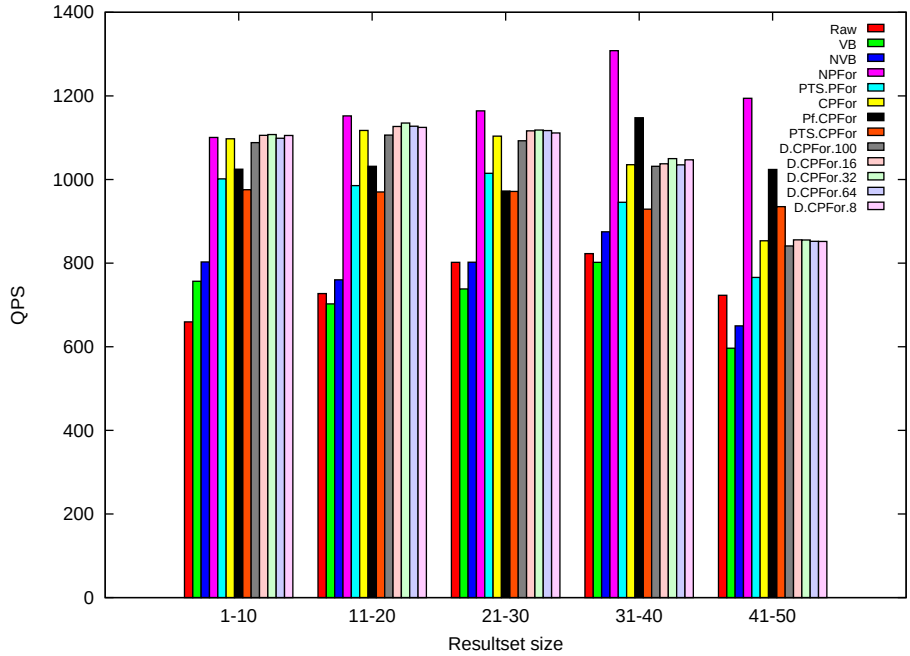

Figure 7.6: Throughput for `sentence`-scope queries at different result set sizes.

For structural queries the performance characteristics is slightly different, the results is shown in Figure 7.6. The number of queries used in each of the result set ranges for this experiment is shown in Table 7.3. As the table shows the number of queries user here is significantly smaller than for the intersection queries. This makes sense, since the structural queries impose stricter constraints on the results, which in turn would lead to smaller result sets, and fewer queries with any results at all. However, by letting the unit of retrieval be a *portion* of a document (Section 5.2.2) the number of results should increase, but not enough to get very many queries with larger result sets it seems.

	1-10	**11-20**	**21-30**	**31-40**	**41-50**
Queries:	11 580	1 604	641	327	198

Table 7.3: Number of queries used in sentence query experiment.

Having so few queries for the larger datasets might reduce the reliability of the results. However, since the same query trace has been tested in a later experiment

with both *document identifier first* skipping and *path types first* skipping more confidence will be provided later on.

One interesting observation here is that contrary to the intersection queries the row-wise *PFor* methods perform as well as or even better than the column wise implementations. The explanation to these improvements is probably due to the granularity of the methods. When looking at what the different queries retrieve one see that a structural query will retrieve a subset of the occurrences in a document, while a intersection query will retrieve *all* the occurrences within a document. This means that the number of dewey codes retrieved for each match might significantly lower than for a intersection query. To retrieve a dewey code in a column wise storage one needs to decode all the columns for that dewey, while in a row wise dewey store, one would only need to decode the row that the dewey resides within. Roughly speaking column wise storage schemes need to decode an *area*, while the row wise scheme only need to decode a *line*. Even though the column wise methods provide better compression and probably better decompression speed on a per-value basis, the granularity might damage performance when smaller parts of the index is accessed.

One would perhaps expect path type queries to be significantly slower than the intersection queries, however, in the two preceding experiments, this is not the case. One reason can be the size of the results, as mentioned earlier, the results of the intersection queries is larger than the ones for the structural queries. Also, the the intersection queries need to access the scope column, if one only was to return the matching document identifiers querying would be much faster.

Also, the skip procedure used to find match-candidates for the structural queries involve not just skipping to the right document identifier, but also to the correct path type, the distance of such a skip will never be shorter than a skip which only skips to the document identifier, and often it will be longer, especially when looking for special path types such as the `company` scope. Longer skips can be processed faster than shorter skips by utilizing block-level skip information more often, additionally longer skips mean fewer skips, which also may cause the queries to be evaluated faster. However, since these queries were `sentence` queries which, will match the path type of more than 99% of all occurrences (Table 6.2) the effects of these longer skips will be very small.

Another possible reason is the queries them selves, there might be some kind of bias in the queries which causes the higher throughput. The queries might contain terms that have shorter posting lists, or some other unknown characteristics which makes the queries fast. This is however, unlikely, especially for the traces with $1 - 10$ results where there are more than 11000 queries.

The intersection queries showed significant decline in throughput as the result sizes increased. This is not the case for the structural queries, in fact, most methods increase in throughput as the result set size increase. Explaining this is hard, however, the cause might be related to the rather low number of queries. There

can be variations within the different query groups themselves. However, the low number of queries also suggest that the entire query trace touches a smaller subset of the index, possibly causing better cache hit ratios and therefore better performance.

7.3.2 The impact of concurrency

Experiments performed with different number of threads gives different results. The test computer has four cores, so it would be fair to assume that the throughput will increase as the number of thread increases towards four. The query trace executed contained 54 267 intersection queries. A plot of the throughput at 1, 2, 4 and 8 threads is shown in Figure 7.7 as is the average latency for the same experiments in Figure 7.9.

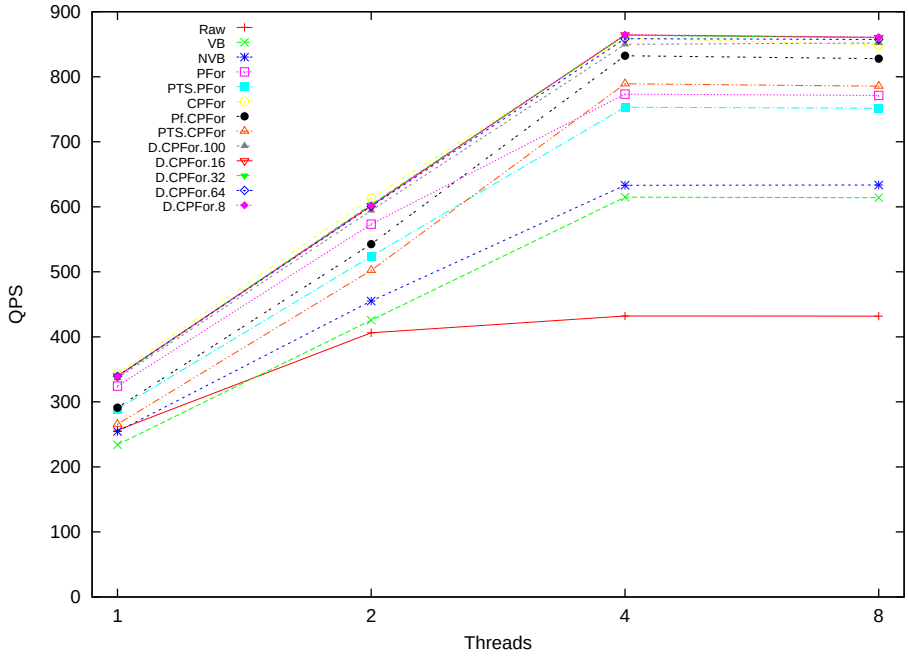

Figure 7.7: Throughput for intersection queries with different number of threads.

It is clear that the uncompressed index suffers from some kind of memory bandwidth problem. As almost all the other methods increase in throughput from two to four threads, the uncompressed version is almost stationary. Similar effects make themselves visible for the simple *PFor* method which is better than both *PTS.PFor* and *Pf.PFor* when running on one thread, while as the number of threads increase to four, all column wise methods beat the row wise methods. These results suggest

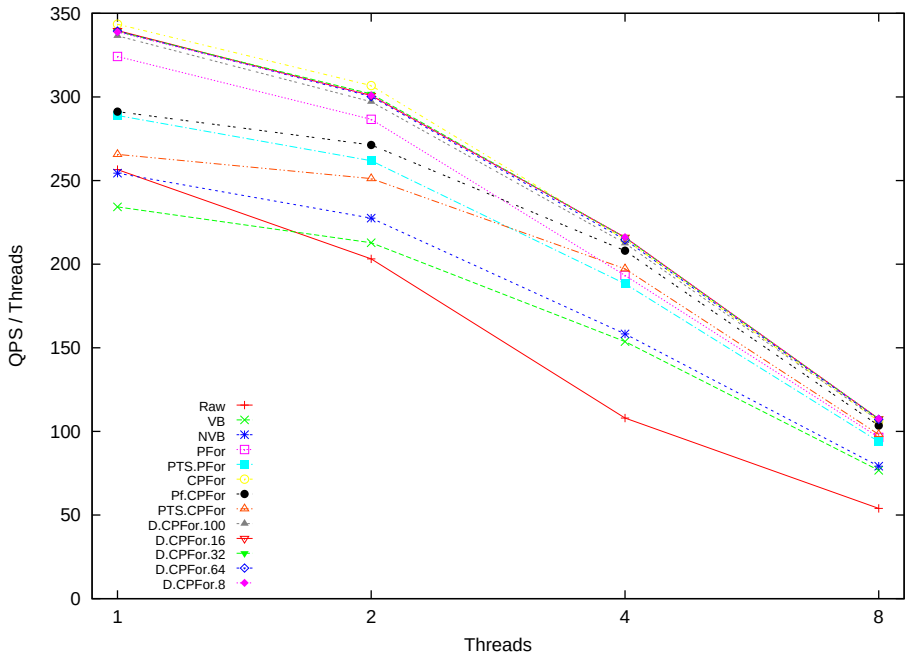

Figure 7.8: Throughput per thread for intersection queries.

that memory bandwidth is an important factor.

The plot in Figure 7.8 shows how the different methods scale with the number of threads. It is clear that none of the methods comes close to linear scaling, however, its clear that some scale better than others. With the current system, there are no need to increase the number of threads above the number of cores. Reasons to why linear scaling is not achieved is first and foremost memory bandwidth, but also overhead related to managing several concurrent threads comes into play.

Latency increases as the number of threads increases- This is as expected and correlates quite well with the throughput per thread results in Figure 7.8. It is clear that when the system is saturated, adding more threads will only add to the latency, and not help on throughput at all. The conclusion from the multi threaded experiments must be that the compressed file size matters, even if there are lots of free main memory space.

Since all the other experiments were run with four threads, the column wise methods have better advantage in those experiments than in the single threaded experiment done in this section. One could argue that this is unfair, and that it does not measure the computational complexity of the different implementation, however, multi core computers is a reality. If multiple threads get higher performance for

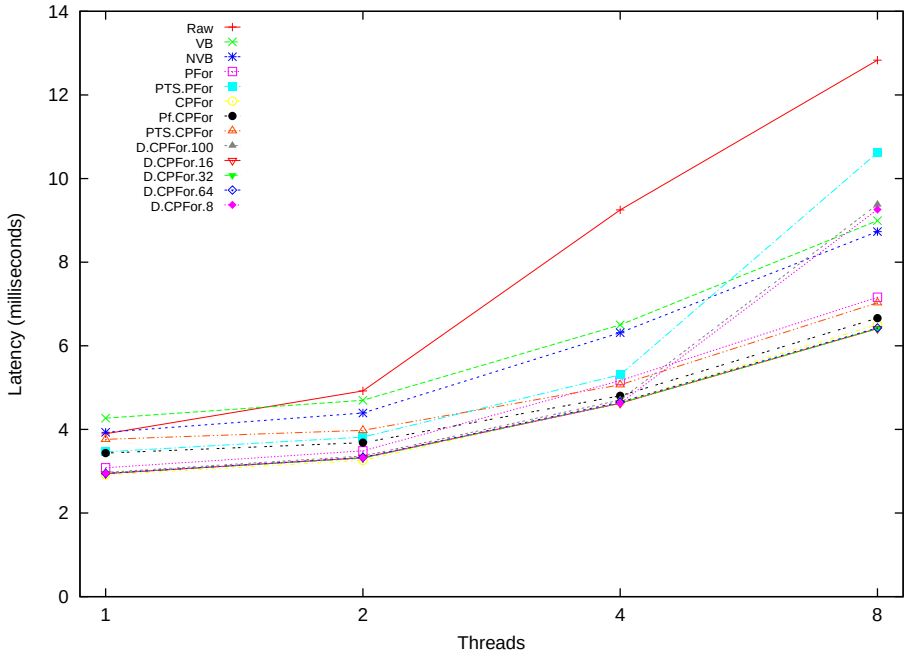

Figure 7.9: Latency for intersection queries with different number of threads.

some method, these methods are better suited on current hardware.

As in the intersection experiments with different result set sizes the *CPfor* and *D.CPFor* variants achieve both the highest throughput and the lowest latency. Suggesting that these methods are the best ones when evaluating intersection queries. And since the performance is so similar one can select the simples, *CPfor*, or the one with the smallest index *D.CPfor*. In any case, further work should evaluate more elaborate ways of flexible tuning the cutoff parameters.

7.3.3 Different skip orders

Experiments with different skip orders and different path types were executed in order to asses how these parameters affected performance of the different methods. If the skip order can impact query performance for a selected subset of the different queries, a query optimizer can be constructed which tries to select the skip method dynamically. However, some of the query types has a rather low number of queries, Table 7.4 shows the different number of queries for each of the path types. It would for instance be hard to draw any real conclusions from the 16 queries which matches `company` scopes. However, the results might suggest that the area can be interesting to look into in further work. If there are enough `person` queries to provide sound results are hard to predict, however, in the experiment with variable scope queries and result set size in Figure 7.6 677 `sentence` queries gave results showing trends close to the experiments with a larger number of queries. Drawing parallels like this is speculative and the confidence in the results from both the `person` and `company` queries cannot be said to be as high as for the `sentence` and `title` queries. Further work should try to get hold of more queries, and possible more data to run them on.

Scope:	**Sentence**	**Title**	**Person**	**Company**
Number of queries:	15 046	1 825	677	16

Table 7.4: Number of queries for the different path types.

In Figure 7.10 the results from the different structural queries executed on different skip-models. What is quite clear from the results that in most cases the skip order does not matter that much. This is probably because both orderings require will end up at the same occurrence. However, there are differences in the `person` and `company` query traces. Here there is actually a negative effect of path type first skipping. It could be that path type skipping is not efficiently enough implemented to get any speedup. It might be that one would need higher level skip summaries than the ones currently implemented in order to make the path type first skip order a viable solution.

Another reason for the failure of the path type first skipping implementation is that the method actually performs path type skipping both first, and last. After the document identifier skip, one need to make sure that the search ended up on a candidate path type, therefore one extra skip is needed. The shear increase of skip operations might explain some of the loss in performance.

A change of skip order will as mentioned above still make the operation end up at the same occurrence. Therefore the possible gain in performance is limited to the speed of the skip operation itself. It could be that implementation effort is spent better in other areas than the skip order. There is probably much more to be gained by optimizing the path type skipping procedures themselves. However,

if further work produce very fast path skipping operations, new experiments with the skip order could be performed to revisit this area.

One reason for the extremely similar results between the skip orders for the `sentence` path type is the fact that 99.92% off all occurrences has a `sentence` path type. Therefore the distance skipped for almost all path type skip operation would be zero. Making the path type skip operation very insignificant for the query processing time.

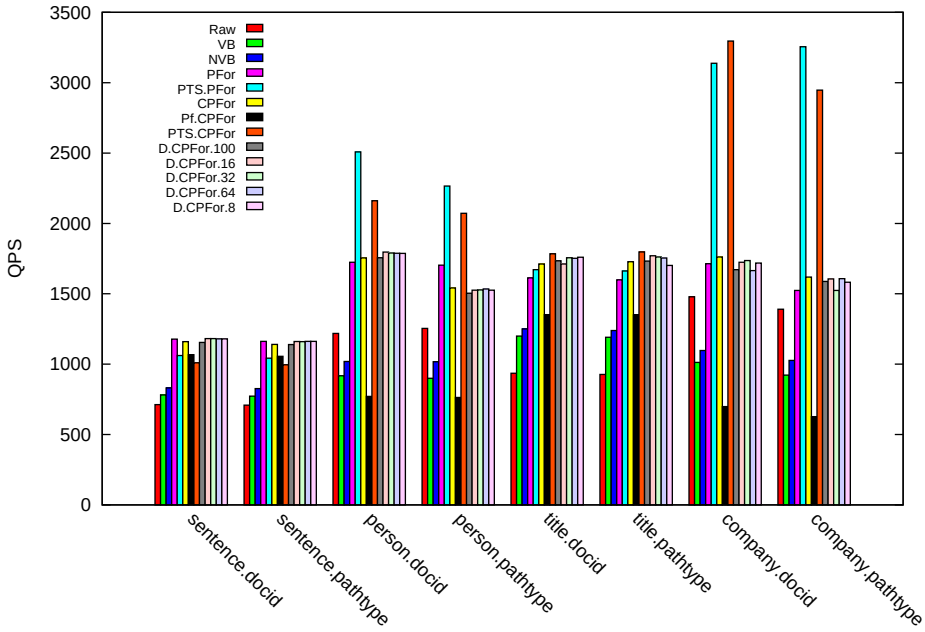

Figure 7.10: Throughput for scope queries with different skip order.

Even though the number of queries is low, it seems that the path type sorted implementations perform very well on the `title`, `person` and `company` queries. This could be both because of more efficient path type skipping due to the encoded summaries, and the ability to decompress a lower number of dewey codes. In order to asses which one of these causes that is the most prominent one, experiments with one of the features turned off could be executed. Further work could try to asses the performance of the different optimizations offered by each method.

One could argue that the number of queries is too low to conclude anything from these results, however, there is a clear trend where the path type sorted methods improve their performance as the selectivity of the path type increases. Even though the confidence to the throughput measurements of 3255 QPS (*Pts.PFor*, *company.pathtype*) with sixteen queries should be low. The results are so much

better that the other methods it should be fair to conclude that the path type sorted methods excel when the path types with good selectivity. In fact when looking at Table 6.2 the results are consistent with the frequency of the different path types.

As pointed out earlier, a small query trace can have better cache performance since most of the working-set might reside in cache memory. This might hide some of the memory bandwidth performance characteristics of the different methods and make more of the computational cost visible in the results. This is supported by the high performance of the *Raw* method in both the person and the company queries. This also suggests that the path type sorted methods have a lower computational cost for those queries. Further work could try to execute a a small query trace many times in order to assess the computation cost of the different methods more precisely.

While the fastest implementations achieved a throughput of 1050 QPS on intersection queries in Section 7.3.1 most method achieve more than 1500 QPS on `title` queries. One explanation which already has been mentioned, is that the skips in each of the lists will be longer than for intersection queries. Therefore the number of skips will be lower as well. This could lead to the higher performance. Also the shear size of the results will be smaller since there are only a limited number of word occurrences within each `title`, `person` and `company` scope.

7.3.4 Managed code versus unmanaged

Changing the implementation of the *PFor* methods from managed to unmanaged code consists of simply turning off the `#pragma unmanaged` directive in the Visual $C++$. The results from intersection queries with both managed and unmanaged code is shown in Figure 7.11. In the following sections $D.PFor$ refers to the version with a cutoff at 32 values. Since the performance of the other versions are very similar those are omitted. Full results are available in Appendix A.

The differences in the results are not enormous, and it seems that the managed code versions fair better when the number of results are lower. However, as the number of results increases, unmanaged code improves. It is hard to predict what the reason for this is, but one possibility is that decoding improves with unmanaged code, and fine grained skipping is hurt by unmanaged code. There exist no concrete evidence of this, however, it is fair to assume that decoding is more computational intensive than skipping, so the cost of making calls to unmanaged functions are amortized over more computations when decoding than when skipping. If one assume that the ratio between skipping and decoding increases as the number of results increase, this could be an explanation. It is, however, purely speculation.

Further work should try to investigate this further, it would for instance be possible

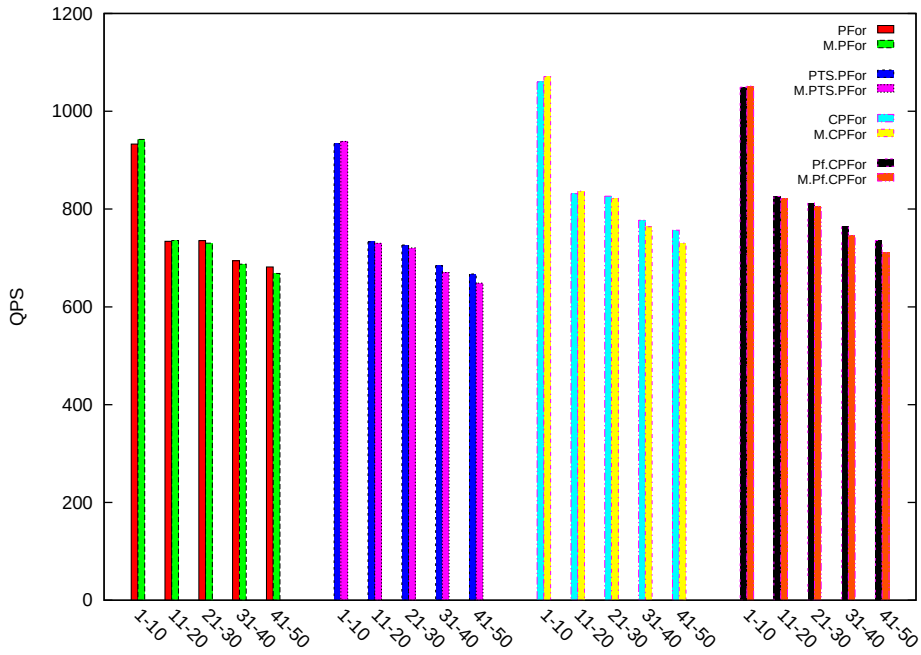

Figure 7.11: Throughput for intersection queries with managed and unmanaged code. Managed versions are prefixed with **M.**

to analyze the concrete cost of such method calls and build some kind of analytical framework for when it is effective to go-native.

The results with both unmanaged and manged code for the `sentence` scope queries are shown in Figure 7.12. For these queries, unmanaged code is always better, and the differences are quite large (approaching 10%), since decompression is just one of many parts of the workload during query processing, this indicate quite a large improvement in shear decompression bandwidth. It also makes it quite clear that decoding is a more important part of the workload of path type queries. This matches quite well with the fact that when processing queries, at least the path type portion of the scope column need to be decoded frequently. Even when it turns out that the chunk was not part of a match, because of the content of a list skipped in later. The intersection queries would not decode the scope column in those cases.

Earlier, various possible reasons for the relatively high speed of the structural queries have been suggested. The reason for the relatively larger difference between managed an unmanaged code given in this section would suggest that structural queries would be slower than intersection queries, since decoding is a larger portion of the workload. It is clear that more profiling work is needed in order to spread

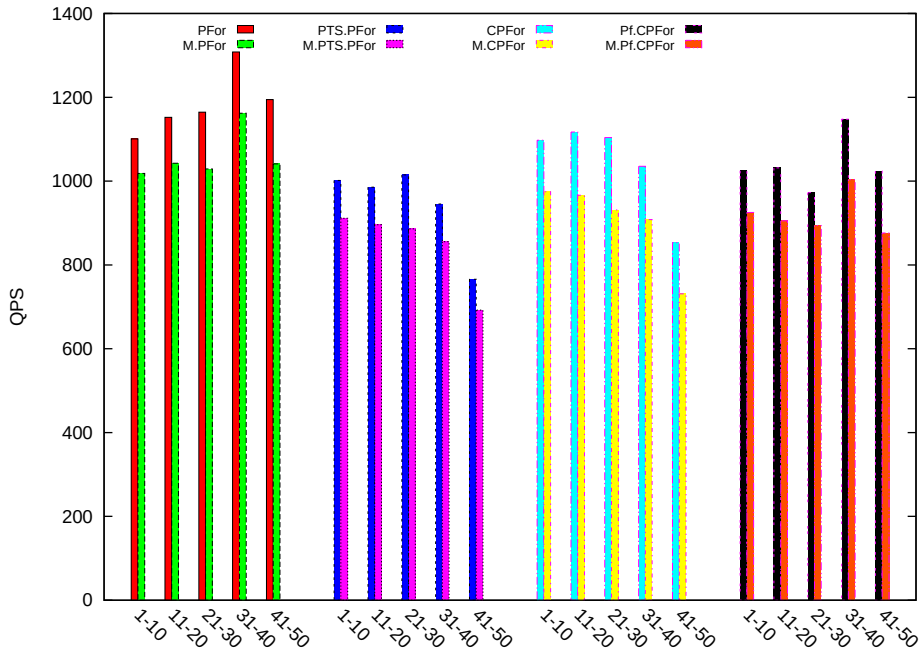

Figure 7.12: Throughput for sentence structural queries with managed and unmanaged code. Managed versions are prefixed with **M.**

more light on where the execution time of the query processing algorithms is spent. The one thing that still is true, is that the results from structural queries contain fewer occurrences, this could in turn be one of the main reasons for a lower total workload for structural queries.

7.4 Critique

The main critique of the experiments is that the number of queries for some of the trails, especially the structural searches were too low. It is for instance difficult to draw any conclusions from only 16 company queries. Also, in [Nat09] some experiments were performed with different document collections. There were some differences in the size of the compressed index, between the different methods. Certainly real search engines need to index different types of documents, or at least documents with different structure. The real world case will therefore be more similar to the case where different collections are indexed. In [Nat09] the path type sorted methods created smaller files than the other methods when a mixed collection was indexed, it would have been interesting to see if the same improvements applied to query performance.

In Section 4.1.3 path type skipping with summaries was described. The only versions using such summaries to speed up path type skipping were the path type sorted methods. This does not give a clear picture of how large the "gain" from adding such summaries can be. To be able to assess this correctly, a pair of methods which are similar except to the summary skipping need to be measured.

Another critique which is well founded, is the lack of some of the more space-optimal compression schemes. For instance *Golomb* or *Huffman* codes. In Section 7.3.2 it is suggested that memory bandwidth is a limiting factor when scaling the execution of queries over several CPU-cores. By testing some methods which create smaller files but spends more time decoding could spread more light on this area.

Also, talking about memory bandwidth without doing low level measurements such as cache hit-ratios include some uncertainty. To support the claims of the query processing with multiple threads being memory-bound one should run experiments with better monitoring the performance counters in both the operating system and hardware.

In order to better understand the performance characteristics of the methods a small subset of the different queries could be run with individual timing. Then the results from the different methods could be used to better understand each methods weaknesses. Such an experiment would probably need to execute each query multiple times in order to be able to record some kind of average performance for each method. One issue then would be caching effects, running one query repeatedly will (if the data it uses fits in cache) remove some of the memory bandwidth issues. One should therefore run some other queries as well in order to let caching effects be more like in some real world.

The explanations of the results obtained suffer from limited insight into where the time is spent. For instance, the reason for the relatively high speed for structural queries combined with intersection queries have not been given with high confidence. It seems that the best explanation is the size of the results returned, however, instrumented executions with detailed timings could help in the analysis.

Chapter 8

Conclusion and further work

This thesis has described the problem of compression and query processing in XML search engines. The focus has been efficient query processing with the compressed index. Extending the work done in [Nat09] methods for both compression, "random access" (skipping) and query processing have been developed and tested. Focus has been on two types of queries, regular intersection queries (AND-search) and structural containment queries (scope-merging). In addition to simple compression implementations as *uncompressed* and *VByte* and *PFor* five different special-purpose compression schemes have been created; column wise *PFor*, prefix coded column wise *PFor*, path types sorted *Pfor*, path type sorted column wise *PFor* and a dynamic *PFor* method. A possible query-optimization technique which manipulates the order of which skip operations are ussed during query processing have been proposed. Additionally implementation details regarding the well known *VByte* compression scheme have been described.

The method has been tested in a minimal search engine test-bench called *NETing*. *NETing* includes an indexer to build the compressed indexes and a query processing framework which handles queries against the compressed indexes. The different compression implementations have been created as plug-ins to this system.

Experiments have been performed with a 382-thousand documents large collection with XML-encoded Wikipedia articles. Queries have been created from the TREC 06 performance tasks. The queries have been processed and filtered and to produce query traces for different experiments. Both throughput and latency have been measured in order to get a picture of the performance of the different methods on different types of queries.

The best performing method on intersection queries is the dynamic *PFor* method which selects between the row wise and column wise storage scheme for dewey codes based on the number of occurrences in the list. On the query trace used

in the experiments this method achieves a throughput ranging from 1100 to 800 queries per second depending on the number of results to the queries.

Running queries in a different number of threads does not scale. The methods which create the smallest indexes such as the column wise methods in general, and the dynamic *PFor* method especially, scales better than those which produce larger files. This suggest that the query processing is memory bound and that further work should try methods which creates even smaller files.

For structural queries with a very common path type, a simple *PFor* method is the fastest one, however, the dynamic and column wise methods come close. However, as the path type becomes less common, path type sorted methods takes the lead. For path types occurring very seldomly, the path type sorted methods are superior to all the other methods. Further work should try to find out if the reason for this: Is it the efficient path type skipping in path type sorted columns, or the ability to only decompress the "interesting" path types? It is probably both, but the path type skipping can easily be applied to other methods by adding some more meta data to the index.

No detectable advantage of executing skips in different order when processing structural queries were detected. However, more efficient path types skipping could make it useful. The idea should not be abandoned, but could be evaluated in further work.

Experiments with the decompression code running both on top of the Microsoft® .NET runtime and as native code were performed. The effects of going native for intersection queries did not present itself as very large. There were a slight advantage for queries with longer results, however, for other query traces, the performance of the managed code beats the native code. For structural queries, the native code performed the best in all experiments, and the differences were large (10%), since decompression is only a part of the query processing work load, the improvements in decoding time must be higher as 10%. This suggests that there in many cases will be quite large gains in going native.

The best choice of compression scheme for the XML-index depends on what kind of queries that is to be answered. However, the dynamic column wise *PFor* method is a very good candidate. It is the best performing method when answering intersection queries, it produces the smallest indexes and it has good performance in most structural queries. The further work section of this chapter will suggest activities which probably will improve this method even further. This include a more dynamic *PFor* method which supports blocks of different sizes. And more meta data encoded in the index in order to support faster path type skipping and therefore more efficient evaluation of structural queries.

When executing queries in multiple threads, memory bandwidth seems to be a limiting factor for achieving high query throughput. Methods which create smaller files therefore have an advantage. It could be that methods which spend more

computation resources during decoding and achieves smaller files can improve performance even more. This should be addressed in further work.

The proposed new *VByte* method also performs better than the one most commonly used in the literature and open source software. Although the improvements are not spectacular, it is easy to adapt to systems already using the *VByte* compression scheme.

8.1 Further work

The experiment results suggest that there are more optimal choices for the cutoff parameter in *PFor* than 100. The optimal choice is probably depending on the data itself. Further work could experiment with this when encoding scopes with *PFor* methods. Clearly, when compressing fewer values *PFor* variants for 32, 64 and 96 items would be more effective than the 128 item variant tested in this thesis. The implementation used in this thesis encodes meta data in a header for each *PFor* block, "selector" info could fit in that header as well and be used by the decoder to choose compression method.

Also, the results from intersection queries show that the methods which has the smallest compressed size of the index scales the best as the number of threads increase. A lot of servers purchased these days have as many as eight cores, so the memory bandwidth issues might be even more important than the effects seen in this tests. The compression schemes tested in this thesis is not the best methods in terms of compression ratio. Methods such as *Golomb* codes and *Huffman* codes can achieve better compression ratios, but have lower decoding speed. By spending more computational resources to consume memory bandwidth the trade offs can be examined more closely. One could even try to compress the lists of different terms with different compression schemes, frequently used terms will for instance have a higher probability of residing in the CPU cache, where the difference between processing speed and bandwidth is different than between main memory and the CPU.

Very many *XML* query processing studies often work with a single document, often called a database such as the Michigan Benchmark [Kan] or the DBLP database [Ley09] to test algorithms answering *XPath* and *XQuery* queries. This thesis is different in that it works with documents more like the "single-webpage-documents" used in web search engines. This makes the document level intersection part of querying the most time consuming task during query processing. By indexing larger documents the part of the query processing algorithms which actually processes the occurrences within each document might get a higher workload. This would make optimizations in dewey matching worth implementing. As mentioned briefly in Section 5.3.4 both prefix coded and column wise methods have interesting optimization to areas which might be more of a hot-spot when querying an index

of larger documents.

The best performing methods in this thesis are the methods which store the dewey code in a column wise manner. Like in column stores, such a layout can effectively deliver projections over the dewey column. One could utilize this to for instance only load some prefix of the dewey codes. Future work could examine if projections of the dewey values could be useful in *XML* query processing.

The native versus managed experiments performed in this thesis used compression code written in Visual C++ and compiled as either managed or unmanaged. Visual C++ could have slightly different performance characteristics than *C#* further work should try to compare managed code written in both languages to each other as well. Additionally, more detailed timings of the different types of method invocations could make a foundation for a analytical framework for choosing when to use native code.

The thesis proposes a branch free implementation of *VByte*, *VByte* is a very popular compression method and any improvements could see widespread adoption. A focused study on the implementation details of *VByte* could be a interesting project. The implementations proposed here has some problems when the numbers to be encoded in small, however, different versions might have better performance for different value distributions.

The experiments in this thesis have only been executed against a single document collection, in real life, different collections are indexed. And also, large collections might have clusters of documents that share some common structure. This could be the case for the Wikipedia collection used here as well, however, future work should experiment with both larger and several different collections. In [Nat09] some experiments with multiple collections were done, the results there suggests that methods such as *path type sorting* improves compression more in those cases.

The results supports a claim that path type skipping can be an effective method to answer *XML*-queries. Therefore, more work is needed on providing efficient path type skipping. The idea of using path type summaries to speed up skipping were introduced in Section 4.1.3, however, more experimentation is needed to assess both the cost in terms of storage, and gains in terms of faster skipping. One could for instance add different types of path type summaries to the dynamic *PFor* method in order to see if it can beat the path type sorted methods on the `person` or `company` queries in Section 7.3.3.

Further more, experiments with a small set of selected queries where the individual timing characteristics would be collected would make it easier to compare performance on a per-query basis. This could aid the development of dynamic methods as described earlier in this section. In one know which kind of queries each method is best (and worst) at one can exploit the findings. Especially if one find types of inverted lists that each method is good at one can dynamically choose to compress that list with the best method. One would of course need an implementation

agnostic index structure and query processing interface like the one in *NETing*.

Even though more queries would give the results more confidence, running small query traces repeatedly will cause the index data to be cached in the CPU cache and executed without the memory bandwidth problem. This would give more exact measurements of the computational cost of the different compression schemes.

In Section 4.2.6 the path type sorting technique is introduced. Several different orderings are suggested, however, only the frequency ordered methods were tested in this thesis. Further work could try to test different variants of path type sorted scope columns. Especially the lexicographical ordering could be interesting to test.

Query optimization is a field of research in the database community [JK84]. The queries executed on the *XML* index could also be subject to query optimizing. It should for instance be possible, even with limited statistics to create a optimization strategy for which skip method to use when processing the structural queries. Further work could design and evaluate such a optimizer. However, even though the results in this thesis did not show any advantage of skipping path types first, improvements in path type skipping could make this worthwhile.

It is also clear that further work should try to evaluate other types of queries, both in terms of *XML*-specific queries, but also ranked retrieval tasks utilizing parts of the indexed data. Further work can extend *NETing* in both its current performance-related direction or in the direction of more features.

Bibliography

[AM04] Vo Ngoc Anh and Alistair Moffat. Index compression using fixed binary codewords. In *ADC '04: Proceedings of the 15th Australasian database conference*, pages 61–67, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[AM05] Vo Ngoc Anh and Alistair Moffat. Inverted Index Compression Using Word-Aligned Binary Codes. *Inf. Retr.*, 8(1):151–166, 2005.

[AM06] Vo Ngoc Anh and Alistair Moffat. Improved Word-Aligned Binary Compression for Text Indexing. *IEEE Trans. on Knowl. and Data Eng.*, 18(6):857–861, 2006.

[BC06] Stefan Büttcher and Charles L. A. Clarke. Unaligned Binary Codes for Index Compression in Schema-Independent Text Retrieval Systems. Technical report, University of Waterloo, 2006.

[BFNE03] Nieves R. Brisaboa, Antonio Fariña, Gonzalo Navarro, and María F. Esteller. (S, C)-Dense Coding: An Optimized Compression Code for Natural Language Text Databases. In *SPIRE*, pages 122–136, 2003.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[BYRN99] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[CCB95a] Charles L. A. Clarke, G. V. Cormack, and F. J. Burkowski. An Algebra for Structured Text Search and A Framework for its Implementation. *The Computer Journal*, 38:43–56, 1995.

[CCB95b] Charles L. A. Clarke, G. V. Cormack, and F. J. Burkowski. Schema-independent retrieval from heterogeneous structured text. In *Fourth Annual Symposium on Document Analysis and Retrieval, Las Vegas, NV*, pages 279–290, 1995.

[Cla99] Clark, James (ed) and DeRose, Steve (ed). XML Path Language (XPath) Version 1.0. `http://www.w3.org/TR/xpath/`, 1999. Accessed: 2010-01-29.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001.

[DHYS08] Shuai Ding, Jinru He, Hao Yan, and Torsten Suel. Using graphics processors for high-performance IR query processing. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 1213–1214, New York, NY, USA, 2008. ACM.

[Eli75] Peter Elias. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, 21(2):194–203, Mar 1975.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[Gol66] S. Golomb. Run-length encodings (corresp.). *Information Theory, IEEE Transactions on*, 12(3):399–401, Jul 1966.

[Gri07] Nils Grimsmo. Faster path indexes for search in XML data. In *ADC '08: Proceedings of the nineteenth conference on Australasian database*, pages 127–135, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[Gus07] Dan Gusfield. *Algorithms on strings, trees, and sequences : computer science and computational biology*. Cambridge Univ. Press, 2007.

[HHMW07] Theo Härder, Michael Haustein, Christian Mathis, and Markus Wagner. Node labeling schemes for dynamic XML documents reconsidered. *Data & Knowledge Engineering*, 60(1):126 – 149, 2007.

[JK84] Matthias Jarke and Jurgen Koch. Query optimization in database systems. *ACM Comput. Surv.*, 16(2):111–152, 1984.

[Kan] Kanda Runapongsa and Jignesh M. Patel and H. V. Jagadish and Yun Chen and Shurug Al-Khalifa. The Michigan Benchmark. `http://www.eecs.umich.edu/db/mbench/description.html`. Accessed: 2010-06-03.

[Knu98] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[Ley09] Michael Ley. DBLP - Some Lessons Learned. *PVLDB*, 2(2):1493–1500, 2009.

[Mic10] Microsoft Corporation. Visual Studio 2010 - Visual C++ - Native and .NET Interoperability. http://msdn.microsoft.com/en-us/library/zbz07712(v=VS.100).aspx, 2010. Accessed: 2010-05-26.

[MRS09a] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval (Online Edition)*, chapter 5, pages 85–107. Cambridge University Press, 2009. Accessed: 2009-10-19, http://nlp.stanford.edu/IR-book/pdf/05comp.pdf.

[MRS09b] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval (Online Edition)*, chapter 6, pages 109–133. Cambridge University Press, 2009. Accessed: 2010-02-01, http://nlp.stanford.edu/IR-book/pdf/06vect.pdf.

[MRS09c] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval (Online Edition)*, chapter 1, pages 1–18. Cambridge University Press, 2009. Accessed: 2010-02-01, http://nlp.stanford.edu/IR-book/pdf/01bool.pdf.

[MRS09d] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval (Online Edition)*, chapter 2, pages 19–47. Cambridge University Press, 2009. Accessed: 2009-10-19, http://nlp.stanford.edu/IR-book/pdf/02voc.pdf.

[MS96] A. Moffat and L. Stuiver. Exploiting clustering in inverted file compression. In *DCC '96: Proceedings of the Conference on Data Compression*, page 82, Washington, DC, USA, 1996. IEEE Computer Society.

[MS00] Alistair Moffat and Lang Stuiver. Binary Interpolative Coding for Effective Index Compression. *Inf. Retr.*, 3(1):25–47, 2000.

[MZ96] Alistair Moffat and Justin Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4):349–379, 1996.

[Nat09] Ola Natvig. Compression for XML search engines, Project report TDT4590 - Complex Computer Systems, Specialization Project. Technical report, Department of Computer and Information Science, Norwegian University of Science and Technology, December 2009.

[OAP+06] I. Ounis, G. Amati, V. Plachouras, B. He, C. Macdonald, and C. Lioma. Terrier: A High Performance and Scalable Information Retrieval Platform. In *Proceedings of ACM SIGIR'06 Workshop on Open Source Information Retrieval (OSIR 2006)*, 2006.

[SAB+05] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and

Stan Zdonik. C-store: a column-oriented dbms. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[SC07] Trevor Strohman and W. Bruce Croft. Efficient document retrieval in main memory. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 175–182, New York, NY, USA, 2007. ACM.

[SCCS09] Haw Su-Cheng and Lee Chien-Sing. Node Labeling Schemes in XML Query Optimization: A Survey and Trends. *IETE Technical Review*, 26:88 – 11, 2009.

[Sha48] C. E. Shannon. A mathematical theory of communication. *Bell system technical journal*, 27, 1948.

[Tro03] Andrew Trotman. Compressing inverted files. *Inf. Retr.*, 6(1):5–19, 2003.

[TVB+02] Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD Conference*, pages 204–215, 2002.

[Vig09] Vigna, Sebastiano. MG4J: The Manual. `http://mg4j.dsi.unimi.it/man/manual.pdf`, 2009. Accessed: 2009-09-29.

[WMB99] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing gigabytes (2nd ed.): compressing and indexing documents and images*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[WZ99] Hugh E. Williams and Justin Zobel. Compressing Integers for Fast File Access. *The Computer Journal*, 42:193–201, 1999.

[YDS09a] Hao Yan, Shuai Ding, and Torsten Suel. Compressing term positions in web indexes. In *SIGIR '09: Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pages 147–154, New York, NY, USA, 2009. ACM.

[YDS09b] Hao Yan, Shuai Ding, and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *WWW '09: Proceedings of the 18th international conference on World wide web*, pages 401–410, New York, NY, USA, 2009. ACM.

[ZHNB06] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-Scalar RAM-CPU Cache Compression. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 59, Washington, DC, USA, 2006. IEEE Computer Society.

[ZLS08] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 387–396, New York, NY, USA, 2008. ACM.

[ZM06] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006.

Appendix A

All query results

	1-10		11-20		21-30		31-40		41-50	
	T	**L**	**T**	**L**	**T**	**L**	**T**	**L**	**T**	**L**
Raw	501.492	7.972	399.698	9.995	402.510	9.900	384.583	10.328	382.530	10.420
VB	768.510	5.198	605.011	6.602	605.932	6.570	569.924	6.947	553.156	7.200
NVB	773.837	5.166	608.900	6.557	610.631	6.512	570.360	6.945	558.280	7.136
PFor	932.876	4.285	734.001	5.439	735.421	5.413	694.343	5.718	681.482	5.855
M.PFor	942.490	4.242	735.654	5.429	730.303	5.457	687.209	5.773	668.263	5.949
Pts.PFor	933.827	4.279	732.939	5.447	726.372	5.487	684.318	5.800	667.317	5.971
M.Pts.PFor	938.358	4.260	729.910	5.470	720.651	5.522	670.186	5.912	648.541	6.136
CPFor	1060.123	3.771	831.523	4.799	826.383	4.823	777.068	5.119	756.823	5.252
M.CPFor	1070.967	3.733	836.648	4.772	821.871	4.852	763.799	5.199	730.415	5.453
Pf.CPFor	1049.309	3.809	825.670	4.838	812.637	4.902	764.258	5.198	735.809	5.406
M.Pf.CPFor	1051.486	3.802	821.295	4.861	805.460	4.942	745.957	5.327	710.426	5.605
Pts.CPFor	1026.553	3.893	803.338	4.972	786.915	5.059	739.897	5.372	705.845	5.635
M.Pts.CPFor	1023.693	3.905	792.653	5.035	771.759	5.158	707.639	5.598	668.730	5.929
D.PFor.100	1057.603	3.780	835.284	4.780	825.598	4.821	775.105	5.124	750.509	5.284
M.D.PFor.100	1078.251	3.707	840.968	4.745	827.652	4.812	766.801	5.175	735.962	5.402
D.PFor.64	1065.779	3.751	835.340	4.782	830.836	4.795	777.291	5.116	754.016	5.274
M.D.PFor.64	1076.861	3.708	838.590	4.761	824.104	4.830	763.716	5.199	732.869	5.437
D.PFor.32	1070.505	3.733	845.030	4.726	835.465	4.765	784.842	5.070	761.421	5.228
M.D.PFor.32	1080.526	3.700	843.899	4.734	828.829	4.812	767.735	5.177	738.377	5.406
D.PFor.16	1073.329	3.725	842.787	4.739	842.445	4.734	786.651	5.019	760.331	5.222
M.D.PFor.16	1082.338	3.693	841.163	4.746	825.458	4.832	769.769	5.155	735.561	5.396
D.PFor.8	1073.416	3.724	845.361	4.724	839.212	4.742	785.338	5.058	760.798	5.221
M.D.PFor.8	1079.993	3.701	845.784	4.721	829.288	4.804	765.922	5.183	732.789	5.428

Table A.1: Full results for intersection queries with a variable number of results.

	1-10		11-20		21-30		31-40		41-50	
	T	**L**	**T**	**L**	**T**	**L**	**T**	**L**	**T**	**L**
Raw	659.381	6.062	727.353	5.336	802.008	4.886	822.802	4.749	723.046	5.208
VB	756.519	5.280	702.459	5.510	738.195	5.231	801.963	4.866	596.574	6.115
NVB	802.914	4.974	760.066	5.063	802.059	4.851	874.941	4.500	649.934	5.562
PFor	1100.742	3.628	1152.230	3.364	1164.353	3.348	1308.211	3.007	1194.392	3.300
M.PFor	1018.415	3.919	1042.752	3.734	1029.146	3.753	1162.035	3.367	1040.973	3.787
Pts.PFor	1001.623	3.990	985.466	3.957	1014.772	3.825	945.576	4.149	765.681	5.108
M.Pts.PFor	911.331	4.382	895.744	4.346	886.187	4.333	855.502	4.579	692.224	5.547
CPFor	1097.578	3.639	1117.303	3.496	1103.832	3.517	1035.377	3.756	853.345	4.644
M.CPFor	974.655	4.095	966.295	4.038	931.173	4.230	908.187	4.263	731.119	5.322
Pf.CPFor	1024.607	3.896	1031.591	3.770	972.594	4.007	1147.853	3.352	1023.910	3.844
M.Pf.CPFor	925.294	4.316	905.476	4.306	894.126	4.377	1003.447	3.821	876.387	4.472
Pts.CPFor	975.742	4.090	970.180	4.014	971.411	4.022	929.177	4.165	934.993	4.146
M.Pts.CPFor	858.856	4.650	828.303	4.711	814.518	4.774	783.885	4.936	632.147	6.105
D.PFor.100	1088.205	3.669	1106.288	3.526	1092.644	3.584	1031.705	3.784	841.056	4.693
M.D.PFor.100	974.418	4.099	959.410	4.062	945.042	4.141	903.490	4.247	731.872	5.338
D.PFor.64	1098.860	3.632	1127.471	3.463	1117.105	3.486	1035.169	3.760	852.282	4.646
M.D.PFor.64	967.685	4.126	956.897	4.071	927.600	4.231	900.464	4.260	728.471	5.384
D.PFor.32	1107.446	3.605	1135.152	3.433	1118.275	3.471	1049.902	3.717	855.262	4.617
M.D.PFor.32	966.024	4.138	956.788	4.080	922.845	4.268	898.458	4.287	726.770	5.409
D.PFor.16	1105.585	3.612	1126.967	3.458	1116.442	3.506	1037.637	3.748	855.843	4.628
M.D.PFor.16	958.699	4.161	948.146	4.109	931.988	4.193	891.439	4.316	722.095	5.456
D.PFor.8	1105.227	3.611	1124.604	3.460	1111.424	3.498	1047.232	3.717	851.881	4.614
M.D.PFor.8	958.857	4.163	951.516	4.096	936.720	4.190	892.847	4.319	722.238	5.466

Table A.2: Full results for `sentence` queries with a variable number of results.

	Number of query threads							
	1		2		4		8	
	T	**L**	**T**	**L**	**T**	**L**	**T**	**L**
Raw	256.587	3.897	406.310	4.920	432.120	9.251	431.927	12.844
VB	234.240	4.269	425.592	4.697	614.873	6.500	613.947	8.999
NVB	254.466	3.929	454.996	4.394	633.101	6.312	633.584	8.730
PFor	324.227	3.084	573.247	3.487	773.243	5.169	771.266	7.162
Pts.PFor	288.913	3.461	523.766	3.817	753.073	5.307	751.456	10.620
CPFor	343.507	2.911	613.574	3.258	859.229	4.652	848.843	7.672
Pf.CPFor	291.196	3.433	542.537	3.685	832.171	4.803	827.685	6.662
Pts.CPFor	265.685	3.763	502.380	3.979	789.050	5.065	785.562	7.029
D.PFor.100	336.528	2.971	594.302	3.364	849.843	4.703	852.036	9.380
D.PFor.64	339.373	2.946	600.463	3.329	858.436	4.656	857.331	6.444
D.PFor.32	339.133	2.948	603.671	3.311	863.517	4.629	860.180	7.567
D.PFor.16	339.732	2.943	601.774	3.322	864.142	4.625	860.337	6.452
D.PFor.8	338.846	2.951	601.876	3.321	864.680	4.622	860.860	9.258

Table A.3: Full experiment results, intersection queries, variable number of threads

	sentence docId		sentence path		person docId		person path	
	T	**L**	**T**	**L**	**T**	**L**	**T**	**L**
Raw	712.605	5.597	708.531	5.634	1218.825	2.752	1253.988	2.666
VB	781.588	5.104	772.459	5.162	917.249	3.389	899.842	3.491
NVB	831.479	4.796	826.742	4.825	1018.698	3.070	1017.400	3.092
PFor	1177.529	3.392	1162.337	3.437	1723.898	1.943	1703.803	1.985
Pts.PFor	1061.687	3.761	1042.403	3.829	2508.064	1.517	2265.628	1.693
CPFor	1160.097	3.440	1140.445	3.500	1754.823	2.018	1541.524	2.274
Pf.CPFor	1066.878	3.744	1055.455	3.784	770.588	3.896	763.519	3.947
Pts.CPFor	1009.409	3.955	995.884	4.008	2161.726	1.681	2072.018	1.891
D.PFor.100	1154.718	3.457	1139.025	3.504	1756.135	1.992	1503.807	2.371
D.PFor.64	1179.969	3.385	1162.148	3.437	1788.242	1.960	1534.375	2.338
D.PFor.32	1181.405	3.381	1160.251	3.442	1790.208	1.947	1527.770	2.330
D.PFor.16	1181.365	3.382	1160.646	3.440	1796.941	1.939	1525.863	2.332
D.PFor.8	1179.446	3.387	1162.200	3.434	1787.050	1.943	1525.585	2.349

	title docId		title path		company docId		company path	
	T	**L**	**T**	**L**	**T**	**L**	**T**	**L**
Raw	935.236	4.249	926.932	4.293	1478.536	1.928	1389.978	2.013
VB	1199.151	3.301	1191.543	3.328	1012.919	2.697	922.375	2.910
NVB	1250.556	3.169	1239.414	3.191	1097.492	2.449	1026.579	2.611
PFor	1613.675	2.464	1599.530	2.489	1713.872	1.636	1523.519	1.783
Pts.PFor	1671.966	2.372	1662.023	2.390	3138.497	0.984	3255.767	0.973
CPFor	1712.330	2.316	1727.651	2.299	1762.195	1.620	1618.502	1.741
Pf.CPFor	1351.904	2.948	1350.125	2.937	698.422	3.901	626.951	4.244
Pts.CPFor	1784.189	2.208	1797.636	2.213	3296.044	0.996	2947.112	1.036
D.PFor.100	1734.423	2.281	1732.761	2.298	1671.955	1.670	1587.712	1.746
D.PFor.64	1752.046	2.260	1754.588	2.267	1665.189	1.670	1607.547	1.739
D.PFor.32	1756.034	2.261	1761.410	2.261	1736.355	1.631	1523.742	1.778
D.PFor.16	1712.296	2.320	1769.771	2.252	1723.967	1.637	1605.311	1.741
D.PFor.8	1759.608	2.250	1701.283	2.327	1718.692	1.649	1581.730	1.757

Table A.4: Full results for structural queries with differet skip order.